



**Filipe José
Jesus Manco**

**Controlo de infraestrutura de rede para Campus
Virtuais**

Network infrastructure control for Virtual Campus



**Filipe José
Jesus Manco**

**Controlo de infraestrutura de rede para Campus
Virtuais**

Network infrastructure control for Virtual Campus

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Rui Luís Andrade Aguiar, Professor associado c/ agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Diogo Nuno Pereira Gomes, Professor auxiliar convidado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor Osvaldo Manuel da Rocha Pacheco

professor auxiliar da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Paulo Alexandre Ferreira Simões

professor auxiliar do Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Prof. Doutor Diogo Nuno Pereira Gomes

professor auxiliar convidado da Universidade de Aveiro

**agradecimentos /
acknowledgements**

Agradeço o apoio da minha namorada, família e amigos.
Agradeço a todos os membros do grupo ATNoG.

Palavras Chave

cloud computing, OpenStack, redes, virtualização.

Resumo

A evolução da actual infraestrutura de rede e modelos de serviço da universidade tem-se mostrado necessária para permitir o fornecimento de serviços inovadores capazes de responder às necessidades do mundo actual. Neste trabalho é proposta uma ferramenta de orquestração de rede que, integrada com a plataforma de cloud OpenStack, é capaz de virtualizar qualquer infraestrutura de rede de forma não disruptiva, proporcionando um modelo de actualização simples dos tradicionais serviços para o novo mundo da virtualização. A framework é capaz de estender as redes virtuais criadas no data-center pelo OpenStack ou qualquer outra plataforma de cloud, para o campus. Fá-lo reconfigurando directamente os dispositivos de rede de acordo com as necessidades, independentemente do fabricante, do tipo de dispositivo ou das suas especificidades, e independentemente da topologia da rede física. O serviço é fornecido ao utilizador usando um modelo de cloud, muito mais flexível que o modelo actual, devidamente integrado com os serviços da plataforma de cloud. O projecto foi desenvolvido com os casos de uso da Universidade de Aveiro em mente, mas o resultado final pode ser aplicado em muitos outros ambientes académicos ou empresariais. A framework é apresentada tanto do ponto de vista conceptual, descrevendo as abstrações e mecanismos criados, como do ponto de vista de implementação, dando ao leitor o entendimento necessário acerca da operação da ferramenta e da sua integração com o OpenStack. A integração deste trabalho com uma visão mais abrangente para o futuro dos serviços da universidade é deixada como trabalho futuro.

Keywords

cloud computing, networking, OpenStack, virtualization.

Abstract

An evolution of the current university's networking infrastructure and service models has been shown to be necessary to enable the provisioning of innovative services that are able to respond to today's needs. On this work a network orchestration tool is proposed that, integrated with the OpenStack cloud framework, is able to virtualize any network deployment in a non-disruptive manner, providing a clean upgrade path from the traditional networking to the world of virtualization. The framework is able to extend virtual networks created on the datacenter by OpenStack or other cloud frameworks, to the outside campus. It does so by directly reconfiguring the network devices according to the needs, independently of the device vendor, the type of device or its specificities, and independently of the specific physical network topology. This service is provided to the end user using a cloud like service model, much more flexible than the current one, properly integrated with the cloud framework services. The project was developed with the Aveiro University use cases in mind, but the final result can be applied in many academic or business environments. The framework is presented both from a conceptual perspective, by describing the abstractions and mechanisms created, and from an implementation perspective, providing the reader the necessary understanding about the framework operation and the integration with OpenStack. The integration of this work with a broader vision for the future of the university's services is left as future work.

Contents

| | |
|---|------------|
| Contents | i |
| List of Figures | v |
| List of Tables | vii |
| List of Acronyms | ix |
| 1 Introduction | 1 |
| 1.1 Motivation | 1 |
| 1.1.1 Scenario: Aveiro University | 3 |
| 1.2 Goals | 3 |
| 1.3 Contributions | 4 |
| 1.4 Document outline | 4 |
| 2 Cloud computing overview | 5 |
| 2.1 What is cloud computing | 5 |
| 2.1.1 Essential characteristics | 6 |
| 2.1.2 SPI model | 7 |
| 2.1.3 Deployment models | 9 |
| 2.2 Cloud market | 9 |
| 2.3 The Service Oriented Architecture | 11 |
| 2.4 Technologies and paradigms | 11 |
| 2.4.1 Overlay networks | 12 |
| 2.4.2 Software Defined Networking | 12 |
| 2.4.3 Resource virtualization | 13 |
| 2.5 Cloud frameworks | 16 |
| 2.6 Fog computing | 17 |
| 3 Fog computing at Aveiro University | 19 |
| 3.1 Motto and use cases | 19 |
| 3.1.1 Scenario 1 | 23 |
| 3.1.2 Scenario 2 | 23 |

| | | |
|----------|---|-----------|
| 3.1.3 | Scenario 3 | 24 |
| 3.1.4 | Scenario 4 | 24 |
| 3.2 | Cloud framework | 24 |
| 3.2.1 | Development testbed | 25 |
| 3.3 | OpenStack platform and ecosystem | 25 |
| 3.3.1 | OpenStack ecosystem and governance | 26 |
| 3.3.2 | OpenStack architecture | 27 |
| 3.3.3 | Neutron: the network service | 28 |
| 4 | System Design | 31 |
| 4.1 | Requirements and major decisions | 31 |
| 4.2 | Conceptual solution and macro blocks | 33 |
| 4.2.1 | Network abstraction | 34 |
| 4.2.2 | Virtual network mapping | 35 |
| 4.2.3 | Device management | 36 |
| 4.2.4 | Network controller | 38 |
| 4.3 | Implementation | 39 |
| 4.3.1 | Software architecture | 39 |
| 4.3.2 | Databases | 41 |
| 4.3.3 | External APIs | 41 |
| 4.3.4 | Command Line Interfaces | 41 |
| 4.4 | Framework operation | 42 |
| 4.4.1 | Physical network management | 42 |
| 4.4.2 | Virtual network management | 43 |
| 4.4.3 | Interoperation with OpenStack | 43 |
| 4.5 | OpenStack integration | 44 |
| 4.5.1 | ML2 integration | 46 |
| 4.5.2 | APIs extensions | 47 |
| 4.5.3 | CLIs extensions | 48 |
| 4.6 | Experiments and the evolutionary view | 48 |
| 5 | Conclusions | 51 |
| 5.1 | Future work | 51 |
| | Appendices | 53 |
| A | Architectural diagrams | 55 |
| B | OpenStack extensions | 63 |
| C | BP: Provider Router Extension | 69 |

| | |
|--|-----------|
| D BP: Campus Network Extension | 77 |
| E BP: ML2 External Port Extension | 91 |
| References | 99 |

List of Figures

| | | |
|-----|---|----|
| 3.1 | Simplified physical network infrastructure backing the exemplification scenarios. | 22 |
| 3.2 | Virtual network described on Scenario 1 | 23 |
| 4.1 | Proposed extensions to the OpenStack data model | 46 |
| A.1 | Class diagram for the Campus Network Controller (CNc) component. | 57 |
| A.2 | Class diagram for the Campus Network Agent (CNa) component. | 59 |
| A.3 | Campus Network Controller Database scheme | 61 |

List of Tables

| | | |
|-----|--|----|
| B.1 | ExternalPort Application Programing Interface (API) attributes | 65 |
| B.2 | API operations for external ports | 67 |

List of Acronyms

| | | | |
|--------------|--|----------------|--|
| AP | Access Point | NAT | network address translation |
| API | Application Programing Interface | NIC | Network Interface Card |
| ATNoG | Advanced Telecommunications and Networking Group | NIST | National Institute of Standards and Technology |
| BIOS | Basic Input/Output System | NFS | Network File System |
| BP | Blueprint | OpEx | Operational Expenditure |
| CapEx | Capital Expenditure | OS | Operative System |
| CLI | Command Line Interface | OVS | OpenVSwitch |
| CNa | Campus Network Agent | P2P | Peer-to-Peer |
| CNc | Campus Network Controller | PaaS | Platform as a Service |
| CPU | Central Processing Unit | PC | Personal Computer |
| CRUD | Create, Read, Update and Delete | PCI | Peripheral Component Interconnect |
| DB | Database | PSTN | Public Switched Telephone Network |
| DHCP | Dynamic Host Configuration Protocol | PTL | Project Team Leader |
| DNS | Domain Name System | REST | Representational State Transfer |
| FOSS | Free and Open Source Software | RPC | Remote Procedure Call |
| GPGPU | General Purpose GPU | SaaS | Software as a Service |
| GPU | Graphics Processing Unit | SDN | Software Defined Networking |
| GRE | Generic Routing Encapsulation | SOA | Service Oriented Architecture |
| GUI | Graphical User Interface | SOAP | Simple Object Access Protocol |
| IaaS | Infrastructure as a Service | SSH | Secure Shell |
| IDE | Integrated Development Environment | SSID | Service Set Identifier |
| IP | Internet Protocol | SNMP | Simple Network Management Protocol |
| ISA | Instruction Set Architecture | TC | Technical Committee |
| IT | Information Technology | UA | Aveiro University |
| L2 | Layer 2 | UML | Unified Modelling Language |
| L3 | Layer 3 | VLAN | Virtual LAN |
| L7 | Layer 7 | VM | Virtual Machine |
| LAN | Local Area Network | VMM | Virtual Machine Monitor |
| LVM | Logical Volume Manager | VPN | Virtual Private Network |
| MAC | Media Access Control | vSwitch | Virtual Switch |
| ML2 | Modular Layer 2 | VXLAN | Virtual Extensible LAN |
| MMU | Memory Management Unit | | |

Chapter One

Introduction

Today's world is IT based. IT is everywhere around us, and we depend on technological services for some of the most basic things we do in our daily lives. University is no exception. If we dig through our academic workflows we find that our universities' IT services are really a basic pillar of our environment. They're used not only to manage our academic path but are also deeply integrated in our learning experience, being it the simple use of web browsing and email, the creation of surveys or the setup of highly advanced research experiments either on technological subjects or in any other field of study, like physics or mathematics.

With all the technological advances our society is witnessing it is fair to expect that the services we are provided with keep up the pace and gift us with state-of-the-art technology that ultimately makes it easier to accomplish our daily duties. But that's not really the case. If we look through the way services are deployed and provided nowadays we'll find an old and ossified infrastructure given to the user through deprecated service models that are unable to evolve and cope with today's demands.

Cloud computing targets this exact problem. This concept redefines how IT services are managed and provisioned, not only for outsourcing models but also for private infrastructure. It has become a game changer on the IT industry, and therefore appears as a very appellative solution for the university's services. But the integration with current infrastructure brings some questions. Completely disruptive solutions are usually not accepted because of the costs they entail, and therefore the integration of the cloud computing domain on the legacy landscape is a requirement for a faster and smooth adoption of this concept.

This work focus on the networking plane. A new framework is proposed to extend the cloud network management mechanisms outside the datacenter, supporting the virtualization of a legacy heterogeneous networking infrastructure, keeping a tight integration with other cloud mechanisms.

1.1 MOTIVATION

Cloud computing is the latest trend on the IT world. It is being widely adopted all around the world because of the advantages it gives for both service providers and its customers.

Although not a new concept, just in the last few years the technology to enable it has been developed, which makes cloud an important and appealing research area.

On its private deployment model, cloud computing is used to deploy services for internal consumption, providing an integrated vision of datacenter management by following a service oriented approach. This paradigm enables an easier and more efficient management of the IT infrastructure and leads to faster deployment of new services. Private clouds are therefore a great opportunity for universities and enterprises to improve and innovate on their services.

However, cloud computing frameworks are mostly targeted at the management of datacenter resources assuming a mostly homogeneous environment. On scenarios consisting of heterogeneous equipments deployed across a broader area, this integrated management is not possible yet. We're talking about fog. Fog computing, as defined by Bonomi et al. in [1], is a new concept that extends the cloud computing paradigm to the edge of the network, focusing in characteristics like low latency, geographical distribution or mobility.

Integration of fog management on cloud computing frameworks would enable innovative scenarios where the cloud, as an entity, would be able to provide not only virtualized services deployed on the datacenter, but also manage an entire campus considering the specificities of each device, such as compute nodes that provide General Purpose GPUs (GPGPUs), and its geographical location, enabling something like booting a Virtual Machine (VM) on the engineering department. Deploying such mechanisms would not only accelerate the evolution of the Information Technology (IT) services, but also ease the management of the IT infrastructure and allow a seamless integration of the cloud in legacy deployments, by providing an integrated management of a mix of virtual and physical devices across an entire campus.

From an economic perspective this vision of fully integrated campus management is extremely important as companies and universities would be able to deploy new and innovative services faster and without the need to replace their hardware, which allied to the use of Free and Open Source Software (FOSS) cloud solutions can greatly reduce operational costs while providing a technological evolutionary step.

Despite its usefulness, cloud frameworks don't currently provide support for this kind of services, but focus mostly on fully virtualized environments, as that is the most common scenario. The integration of fog is a large problem touching multiple areas of the cloud computing ecosystem, namely lower level services like compute, network and storage, but also raising some higher level questions like time allocation for physical resources or resource usage policies among many others. Therefore it is necessary to propose new mechanisms that enable this integration. Section 1.2 introduces the area of actuation of this work, and describes the goals it aims to achieve contextualizing in terms of the problems stated above. But before, our scenario.

1.1.1 Scenario: Aveiro University

Aveiro University currently hosts more than 30 departments across the campus plus two off-site poles directly connected to it. The IT infrastructure comprehends two datacenters hosting more than 200 servers (both physical and virtualized) for research and internal services, 4500 Personal Computers (PCs) serving offices, classes and research, plus thousands of intermittently connected personal laptops and smartphones. IT services run on top of a wired network stretching over the entire campus using a mix of optical and Ethernet technologies. On top of that, more than 450 Access Points (APs) provide wireless access to the clients.

sTIC, university' IT department, is responsible for running a big portfolio of mostly stable internal services, but also support an ever-changing set of requirements, resulting from the high research activity taking place in the university. However most of infrastructure management is still done manually by the technical staff leading to an huge amount of man-hours devoted to routine tasks and small fixes and a lack of flexibility and scalability, limiting research activities, limiting evolution and increasing operational costs.

A new model is then necessary that frees sTIC staff from the burden of routine tasks and provides advanced services to the users enabling the deployment of advanced scenarios without sTIC intervention but keeping the necessary isolation. Ultimately, the objective is to provide a fully virtualized campus that goes from the datacenter to every classroom.

Although cloud technologies are starting to be adopted with the purpose of solve some of the problems, as already noticed technology to implement the next generation campus network, based on the vision of a fully integrated management presented before, is still missing. This technology is the key requirement for a better service provisioning and to enable faster and richer innovation.

1.2 GOALS

On the previous section a vision of a fully integrated environment was presented with all the advantages it provides. It was also noticed that cloud frameworks are not yet targeted at the use cases envisioned, missing some key functionalities. But fully implement the envisioned scenario would be a very extensive work, therefore this dissertation focus on a specific subset: the cloud networking plane.

From an high level point of view, this work aims to extend a cloud framework with mechanisms to support the automatic management of a physical network and integrate its resources in the cloud virtual network environment. This means that, through a centralized management point, a user will be able to fully operate a mix of virtual and physical networks and interconnect them.

The mechanisms should be integrated with Neutron, the OpenStack Networking platform. OpenStack platform was chosen for a couple of reasons, amongst which the following are

highlighted: (1) is an open source solution, (2) is currently the most popular FOSS cloud framework, (3) has a very active community developing it and (4) was designed to be extended.

Despite the technical details, the work developed must fulfill some base requirements:

- Centralized interface: OpenStack interface should be extended to support new operations;
- On demand provisioning: all network configurations should be applied automatically and on demand;
- Vendor independent: the extensions developed should be easily extensible in order to use any network equipment from any vendor;
- Non disruptive: the new solution should work on top of an already running system, without service disruption, i.e. the framework doesn't have full control of the equipments, and shouldn't require operations like resetting configurations or rebooting devices.

1.3 CONTRIBUTIONS

This work resulted in the submission of three Blueprints (BPs) for the OpenStack project. The first BP, named “Provider Router Extension” [2] (appendix C), details how Neutron should be patched to enable the usage of physical routers in addition to its virtualized counterparts. The objective of the BP is to provide a simple way to implement some of the scenarios proposed for this work. The second BP, named “Campus Network Extension” [3] (appendix D), describes the solution resulting from the work developed on the context of this dissertation. Finally, the third BP, named “ML2 External Port Extension” [4] (appendix E), was requested by the community to detail how the changes proposed by the “Campus Network Extension” could be integrated in the new “ML2” plugin developed during the Havana development cycle (released in October 2013).

1.4 DOCUMENT OUTLINE

The remainder of this document is organized as follows: on chapter 2 cloud computing is defined and explored from different perspectives (evolution, market and technology); on chapter 3 the Aveiro University (UA) use case is described using scenarios, together with the major goals of the project. OpenStack is also introduced here; on chapter 4 the proposed solution is described in detail both conceptually and from an implementation perspective; finally, on chapter 5 the main conclusions are drawn and the future work is described.

Chapter Two

Cloud computing overview

What is and what is not cloud computing has been a very hot topic in the last few years. Whether cloud is an ordinary evolution of the IT services or a completely new technology, or even if cloud is really a technology rather than a complex business model, were open topics for quite some time. Fortunately, today's companies and the community in general are getting to a definition of what cloud computing really is and a formal definition of cloud computing, the only formal one at the time of writing, was published by National Institute of Standards and Technology (NIST) .

In this chapter cloud computing is introduced with some historical background. The cloud market is discussed and the technology behind cloud is explored.

2.1 WHAT IS CLOUD COMPUTING

Before digging into what cloud computing really is, it is useful to think about the evolution of computers and, on a broader sense, the IT services, mainly how they were used and provisioned over time.

It's undeniable that the born of computers was a great step to the present society. But the way society has taken advantage of computers, and the business models around them has clearly changed over time. We get from the mainframe model, where a main central computer was serving multiple users through thin clients, to the personal computer, the client server based model and web based solutions, everything with a single purpose: make computers more available, useful and profitable. Although cloud can be revolutionizing the market, the concept behind it was already being applied much before the cloud hype, in the terms of IT outsourcing. IT outsourcing consists in obtaining IT related services from a third party that is responsible for managing the infrastructure.

As defined by NIST in [5], cloud computing is “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”

2.1.1 Essential characteristics

Following the NIST document, a true cloud computing service counts with five essential characteristics. Let's take a look at them and understand what they mean and how they influence the business model.

“On-demand self-service”. Resources can be requested or released with minimal provisioning time and without human interaction with the provider.

Giving customers the power to unilaterally and automatically define the exact amount of resources they want to use at each point in time let them eliminate the over provisioning problem and consequently reduce costs while keeping a scalable environment. From the point of view of the provider this characteristic brings two main advantages: it reduces the burden associated with service provisioning as done in old models, where human operations were necessary, and it makes pay-per-use models very attractive for customers, since they decide exactly what resources they use and how.

“Broad network access”. Services are provisioned over the network and accessed through standard mechanisms like web-browsers and standard APIs.

This means customers will be able to manage cloud services simply by using a generic laptop or even a smartphone, needing only an Internet connection. Even non technical people will be able to manage some of the cloud provisioned resources giving enterprises a great flexibility and business advantage.

This ubiquitous access to IT resources is also a step forward on the democratization of IT services, by giving small enterprises a solution to use enterprise graded resources, without needing expensive and proprietary solutions.

“Resource pooling”. Provider resources are pooled to serve a multiplicity of clients on a multi-tenant scheme, i.e. the same hardware can host different services for different customers (tenants) on a transparent manner.

The concept of multi-tenancy is great for both customers and service providers. From one side, customers ignore the complexity of the real datacenter having the idea of being use dedicated resources with guarantees of isolation from the other customers. For providers this means a better use of resources since hardware stops being dedicated to a specific costumer (which typically leads to underutilization of resources), and therefore being more efficient and profitable.

“Rapid elasticity”. The provisioned resources rapidly scale up or down according to customers' needs and appear to be unlimited at the eyes of the clients.

Together with the first characteristic, this lets customers build really scalable environments that fit any business needs. Cloud should be able to support anything from a very stable to completely unpredictable requirements.

“Measured service”. Cloud providers offer metering capabilities along with the provided resources.

The monitoring of the used resources give tenants a competitive advantage by letting them not only control costs, but also being able to accurately predict usage patterns and adapt to them.

2.1.2 SPI model

As explained before, cloud computing is not a service or technology per se, but rather a paradigm for provisioning resources as a service. Multiple different resources are valid clients to be supplied under the “as a Service” model, ranging from raw hardware to complex software. As cloud computing evolved the provided resources were rapidly stratified into three layers denominated service models, each providing resources of a different level of complexity.

On the lower layer we have Infrastructure as a Service (IaaS) providing raw computational resources like Central Processing Unit (CPU) cycles or memory. On top of the infrastructure a second layer provides a Platform as a Service (PaaS) service model orchestrating the raw resources provided by the lower layer to provide complete and ready-to-use development environments. Finally on the top of the pyramid Software as a Service (SaaS) is the user facing layer serving end-user software built on top of the previous substrate.

This three layered stack was denominated “SPI model”, SPI obviously standing for Software, Platform and Infrastructure. As we can easily see, these layers are not mutually exclusive, but rather built on top of each other, and as such, the SPI model is also sometimes referred to as the cloud computing Stack.

Lets now take a look at each service model: what it provides, how it can be used and how they interact with the other models.

The infrastructure layer provides all the fundamental computing resources we found on a datacenter, from CPU, memory and storage to basic network connectivity and advanced networking services. It can be seen as providing an all datacenter on a service model, where customers can then build their specialized environments to deploy their applications with all the freedom to choose the appropriate architecture run any operative system and deploy any software as they would do in a dedicated datacenter. In fact IaaS can basically provide the costumer approximately with the same freedom as dedicated hardware without having to buy and maintain it, plus enable the infrastructure to scale up or down according to needs.

The IaaS model is useful for customers needing to build very specialized environments, for customers needing a fine grained control over the used infrastructure, or something on this line.

Although from the point of view of the user the resources appear to be fully dedicated, under the IaaS model, the costumer doesn’t usually have direct control over the physical datacenter infrastructure but rather controls a virtualized environment running on top of the raw resources. IaaS relies on virtualization mechanism, making easier to implement the multi-tenant model promoted by the cloud computing paradigm.

On top of the basic infrastructure some providers build orchestration tools and deployed full featured, ready-to-use development and deployment environments where customers can easily build and deploy their software products without requiring the usually complex setup of all the necessary development and deployment machinery.

We're talking about PaaS. Comparing it with IaaS, instead of providing VMs and networks, PaaS providers give tenants, Integrated Development Environments (IDEs), compilers and debugging tools, software frameworks, databases, web servers and a broad panoply of resources all ready-to-use with minimal configuration and maintenance for the end-user. This lets customers focus on the effective development of the software instead of all the underlying infrastructure.

Finally, on top of the stack we have SaaS. On the SaaS model, customers don't build anything. Providers give ready-to-use software that clients can use, without install, update, backup, or in any way maintain any infrastructure that is providing the software. Software is delivered over the network and maintained by the service provider.

This is the most know model of cloud computing, because it is faced at the end-user, either business users, or particulars. Obviously, SaaS solutions may be built on top of the platform and infrastructure layers, but that's not a requirement.

This three layered model is now generally accepted being defined on the NIST document. Regardless this isn't yet black and white.

If we carefully look at the services provided under each service model we can easily see the line between them isn't that sharp. What happens if an IaaS provider starts offering operative systems as part of its service? Should operative systems be part of the IaaS model or of the PaaS. There are also some providers giving tools to ease the orchestration of IaaS resources. Should these tools be part of an IaaS offer? Looking now at the PaaS level. An IDE is software after all, shouldn't it be part of a SaaS solution? And a team management application, that is part of the software development process, should it be PaaS or SaaS?

The examples above serve to give the idea that although the terms are now generally accepted by the business stakeholders and community in general, these terms evolve over time as technology evolves, and if we add the business hype it is very difficult to accurately define each one of them.

If we look through the "cloud-o-sphere" we may find lots of different "*aaS" offers, like Database aaS, Security aaS, email aaS and many other, that is worth discussing. What happens with all these names is mainly marketers trying to use the last buzz words to promote their product. But if we carefully look to each "*aaS" offer we can usually see they are particular cases of the service models defined above. Although they represent valid use cases of the cloud, they aren't really individual service models, at least according to the current definitions we may find. Nothing prevents one of these to become a service model in the future as the cloud itself evolves and new services and technologies appear.

2.1.3 Deployment models

Now that we have discussed service models, and the SPI stack, we can analyze the different deployment models for cloud computing solutions. NIST defines four models to deploy cloud services basically based on three main characteristics: who owns and operates the infrastructure, how the infrastructure is shared among tenants, and for whom the service is available.

The first model is the Public Cloud. Under this model, cloud services are accessible to the general public and the cloud infrastructure is solely managed by the provider, resources are shared among the customers on a multi-tenant model. This is probably the most well know and most used type of cloud and is sometimes considered the only true cloud model because it leverages the primary objectives of cloud computing. This topic is further discussed on section 2.2 where the business around cloud is analyzed.

Contrasting with the public model we have the Private Cloud. On this model resources are allocated to a single entity and are only available to that entity. The infrastructure can belong and be operated in house, or can be outsourced by some cloud provider like in the public cloud model.

Although the private cloud model eliminates some of the benefits of cloud it may be a good solution on some situations.

The next, Community Cloud model, is a mix of public and private cloud. This model is very similar to the private cloud, but the services are available to a community that have shared interests instead of only be allocated to a specific entity.

Finally we have the Hybrid Cloud solution. This last consists in using a composition of two or more services from the three different types presented above. The different solutions should be orchestrated to provide a single service on this model.

Notice that, although cloud is mostly about business, none of the previously mentioned models necessarily relates to business. What is important to distinguish in the different deployment models is how the services are provisioned and the resources shared. It is perfectly possible to have cloud providers and customers without business interests.

2.2 CLOUD MARKET

After a brief introduction of the basics of cloud computing it is interesting to understand how cloud affects business. In fact this work is at least partially about business given the goal is to leverage existing infrastructure enabling innovation while reducing costs.

As already discussed, cloud is the new paradigm for computing services provisioning that is changing how enterprises provide and consume IT services. It is a new breed of outsourcing models that brings advantages not only for customers but also for providers. It is therefore natural that enterprises are moving their systems to cloud computing providers. But how is that affecting business? What are the advantages and disadvantages? Are there any problems related with using cloud services? Those are the questions we're going to look at now.

One of the major problems with the traditional IT infrastructure is the capital investment necessary to deploy, run and upgrade a service. It is necessary to buy hardware and software, maintain expensive facilities, account redundancy, security, backups and catastrophe recover, and then pay to a team of professionals only to keep the systems running.

The high costs of maintaining on-premise infrastructures are prohibitive to many small enterprises and are a limiting factor for innovation on bigger ones. Cloud solves this problem by eliminating the necessity to own anything, and consequently eliminating the usually necessary initial investment, representing a shift from Capital Expenditure (CapEx) to Operational Expenditure (OpEx) .

Predicting the infrastructure necessities is another big problem related to traditional IT models. Enterprises either consider the worst case scenario and as such always over-provision resources leading to a waste of capacity or may not be able to attend demand, both leading to waste of money. Cloud is elastic, meaning that you can request resources as you need and release them as soon as they aren't necessary anymore. Together with a predictable pay-as-you-go model, cloud let enterprises reduce costs and eliminate the necessity to carefully and accurately estimate future infrastructure necessities.

Moving to another point, Gartner says in [6] that enterprises are spending 80% of their money on managing their IT infrastructure. This is a huge point to consider when discussing traditional IT. With traditional IT enterprises spend much money on non-strategic activities that bring no advantage in the market. Moving to cloud enterprises can get a larger focus on the core business leaving IT management to specialized enterprises, i.e. the cloud providers.

Until now we have been looking at the direct advantages of cloud computing. All of them assume a deployment model with off-premise infrastructure, managed by a cloud provider. But as we have seen on section 2.1.3 there are deployment models that consider on-premise cloud infrastructure managed by the same entity that consumes the resources, some enterprise in this case. This models appear to defeat all the advantages that cloud computing provides, so why are they used? To better understand lets first look at the problems related to the use of cloud computing.

Security is one of the most frequently mentioned concerns when an enterprise is considering a move to the cloud. With cloud computing, enterprises are moving their away from their full control, possibly posing security risks on confidential data. It may indeed be possible that enterprises can't use cloud computing because of legal constraints.

Another big problem related to cloud computing is the risk of vendor lock-in. Most of the times, cloud vendors provide proprietary frameworks and APIs that let customers locked to that particular solution, unable to move to another provider or use multiple providers at the same time. Although this is one of the problems of cloud computing it already happens with traditional IT solutions.

Finally, another important aspect to consider is performance. Although most of the enterprises will be fine with cloud solutions, enterprises having less usual constraints and requirements for their IT solutions (e.g. executing resource intensive tasks) may not be well

served by these solutions.

That's where the secondary deployment models come in. Although some of the benefits of cloud may be lost by using a different deployment model, private and mainly hybrid models may be a good solution for the problems stated above.

Summarizing, the final goal of any business is to generate value. Despite of not fitting everywhere, cloud changed the business and it is undeniable that it's here to stay, making it a great research area.

2.3 THE SERVICE ORIENTED ARCHITECTURE

But what's the big innovation of cloud computing? If we look through the available offers we can't find anything technically revolutionary. Server virtualization is here for years, the same with storage. Network virtualization is new but is not mandatorily associated with cloud. Authentication and authorization services are also available for years. And the same for all the other services, so technologically speaking there is no big revolution on cloud computing. The revolution is all about how the service is provisioned. And there is a very important software development paradigm that fits perfectly in the cloud computing world: Service Oriented Architecture (SOA) .

SOA is a software design paradigm with a focus on services. A service being a self-contained software entity with very focused and restricted functionality, exporting a well defined interface. Applications using the SOA architecture are then basically a collections of articulated services.

Although it is possible (and in fact it happens), to develop a cloud computing framework without the SOA paradigm in mind, cloud frameworks could greatly benefit from this idea, since the cloud philosophy perfectly matches the SOA principles. Cloud is also a collection of articulated services that, altogether provide a useful application to the final user. Therefore using the SOA principles when design a cloud computing solution will certainly make it more scalable, and more useful, since developing an application is in fact developing a set of independent services that can then possibly be used outside of the original context.

The OpenStack solution that we present bellow and discuss on chapter 3, makes use of the SOA principles. So it would be useful to have this in mind when analyzing that solution.

An important aspect of the SOA paradigm is how the interfaces are provided. Although there is no big constraint on how to publish the interface, state-of-the-art solutions, usually use Representational State Transfer (REST) , Simple Object Access Protocol (SOAP) , or any derivation of these two. What this allows is the quick development of new services, since these frameworks are already known and there is a lot of source code and documentation about it.

2.4 TECHNOLOGIES AND PARADIGMS

On this section we will introduce some concepts, paradigms and technologies that will be useful as we move through the document.

2.4.1 Overlay networks

An overlay network is basically a logical network built on top of one or more networks. The overlay network, although dependent on the substrate infrastructure runs in an independent manner.

There are multiple examples of overlay networks. The internet itself began as an overlay network on top of the Public Switched Telephone Network (PSTN). The telephone network was only used to carry packets across the endpoints, everything built on top of that, namely the Internet Protocol (IP) is completely independent of the used infrastructure. Peer-to-Peer (P2P) networks are also overlays built on top of the internet, using different mechanisms to address the endpoints and to manage the connectivity, also with a different objective, but using IP as the backend.

Advantages of the overlays are mainly relative to the provided isolation. Using overlays we can build isolated networks, possibly using different technologies overlapping address spaces, etc., without having an independently infrastructure.

On a cloud computing world, overlays are what enables providers to create isolated virtual networks for its tenants.

2.4.2 Software Defined Networking

Originally, computer networks were designed on such a way that both data and control planes were deployed together on the networking devices. Software Defined Networking (SDN) is a new paradigm claiming that the control plane should be separated from the actual packet forwarding and centralized on a separate device, to improve performance and flexibility.

To better describe the idea let's look at how things work on a switch. Traditionally switches forward packets based on rules constructed internally by the switch firmware, using some learning mechanism. All the brainpower (or lack of it) is on the devices themselves. What SDN proposes is that the rules are created centrally on the network controller and then injected on the devices, so their unique responsibility becomes forwarding traffic.

There are multiple problems with the traditional method. To attain a good performance network devices need to implement the control plane in hardware, which causes two major problems: (1) devices become either too simple and limited in functionality, or too expensive when providing extra capabilities, but in either case (2) devices are completely inflexible in terms of evolution, since everything is deployed on immutable hardware. These two problems are serious constraints to the evolution of the networks leading to the ossification problem we're assisting today.

By moving the control plane away from the network devices, these become simple packet forwarding machines, which extremely simplifies the hardware, reduces prices, and enables data plane optimizations previously impossible. Having a central control entity also brings an unprecedented flexibility to the network, by enabling administrators to easily deploy new technologies and paradigms without having to change the network devices.

SDN is only a concept, that may vary deeply among implementations. Currently, the most used SDN solution is OpenFlow [7].

2.4.3 Resource virtualization

One of the properties of cloud computing is multi-tenancy. Multi-tenancy refers to an architecture where the same software instance is able to serve multiple clients concurrently and on isolation, as opposed to single-tenancy where each client has its own dedicated instance. On the context of cloud, the meaning of the multi-tenant architecture has been extended to the hardware, meaning that, on a multi-tenant environment, a single hardware instance, like a server, is able to partition its resources providing an isolated slice to each one of its tenants. Discuss how an hardware resource is partitioned and each partition isolated in order to create an efficient IaaS cloud is the point of this section.

We're talking about virtualization. Virtualization is a broad term that covers a multitude of technologies with different goals, but in simple terms it can be defined as the creation of a virtual (as opposed to real) version of some service or resource, by decoupling the service from its physical realization.

The concept of virtualization was first used in the mainframe era, back in the 60's, to describe the partitioning of large mainframes into multiple logical servers. The system, developed by IBM and documented in [8], allowed engineers to use each logical (virtual) server as an independent machine, possibly running different operative systems on top of the same physical mainframe. The concept was very successful but was somehow forgotten with the advent of inexpensive x86 servers. Already in the 90's, VMware adapted the concepts introduced by IBM to the x86 platform, developing the first hypervisor able to run on top of this type of processors. This technology was fairly accepted mainly because it led to a much more efficient use of resources, reducing the number of physical servers necessary on a datacenter. This technology are the base of the most well know type of virtualization these days: the server virtualization. But, with the evolution of technology and the increase of market demands, the concept of virtualization was recently expanded to almost every part of an IT infrastructure, from servers, to storage, network or even applications. The evolution was not solely an evolution of technology but also an evolution of the concept of virtualization mixed with the technology hype. Partitioning an hard drive fits in the concept of virtualization. Although some of them may be questionable, in [9] 8 different forms of virtualization are presented, what shows the wideness of the concept.

On the context of this work we're mostly focused on two forms of virtualization directly applied to the IaaS cloud model: server virtualization and network virtualization. This types of virtualization are discussed in the subsections bellow.

Server virtualization

Server virtualization is nowadays the canonical example of virtualization, but it can be simply defined as the creation and execution of VMs. A VM is an abstraction layer that sits somewhere above the hardware to create isolated partitions of the underlying resources

(e.g. CPU, memory, storage, etc.) and presents them as if they were part of a real (physical) machine, on top of which an Operative System (OS) , called the guest OS, is executed, decoupled from the real hardware. The implementation of the abstraction layer is usually done through a piece of software called hypervisor or Virtual Machine Monitor (VMM) . How these hypervisors are executed and the functionalities they provide depends on the type of virtualization implemented and the type of hypervisor we are referring to.

We can find multiple flavors of virtualization. If we look through the literature we can find a lot of different concepts, sometimes referred as different types of virtualization. Although each one has their benefits (and also shortcomings) they usually share a common line and differ only in some characteristics.

Full virtualization is a type of virtualization where the all the hardware interfaces are emulated by the hypervisor in such a way that the guest operative system isn't aware that it is being virtualized. This is the type of virtualization that provides the higher level of compatibility with guest operative systems that can run unmodified, but is also the least efficient one since it adds a heavy and complex layer of software between the OS and the hardware.

With full virtualization, every aspect of the hardware must be emulated. The privilege instructions of the CPU Instruction Set Architecture (ISA) , the Memory Management Unit (MMU) , Peripheral Component Interconnect (PCI) buses, hardware interrupts, Basic Input/Output System (BIOS) , and every other aspect of the hardware must be carefully emulated and present to the guest OS, what represents unnecessary overhead most of the times. A good example is the emulation of Network Interface Cards (NICs). Being virtualized, the guest OS could simply drop application traffic into a buffer to be then properly processed by the hypervisor before being delivered to the physical NIC. However the guest OS is running the physical NIC drivers, following every detail of the communication with the physical hardware to talk with an emulated version, which is clearly less efficient.

Another type of virtualization that doesn't suffer from this problem is paravirtualization. Using this type of virtualization the guest OS knows it is being virtualized and, when necessary, communicates directly with the hypervisor through what's called hypercalls, that allow a direct communication with the hypervisor instead of using privilege instructions that would have to be trapped and emulated. Device drivers are also special drivers that communicate directly with the hypervisor, instead of the traditional ones. But this mode also has some limitations, mainly because some hardware facilities are not available to guest OSs.

The paravirtualization concept born from the difficulty in virtualize certain platforms like x86, given that, among other problems they don't comply with the formal requirements for virtualization, as defined in [10]. With the advances in server virtualization, hardware vendors started to embrace the concept and provide virtualization extensions to its processors, mainly Intel with its Intel VT technologies [11], and AMD with AMD-V technology. This new technology gave raise to a new type of virtualization called hardware assisted virtualization. This technology greatly simplifies the hypervisor work and improve the performance of the

virtualized systems by providing specific instructions and facilities to run VMs.

The differences among these types of virtualization are discussed in the white paper from VMWare. Beyond this three types of virtualization we can also find some other hybrid virtualization modes that try to get the best from both worlds, like the technique described in [13].

Concerning hypervisors we can distinguish two types: type I (also called native or bare-metal hypervisor) and type II (also called hosted hypervisor). Type I hypervisors runs directly on top of the hardware, booting before any Operative System, contrasting with type II hypervisor that sits on top of an host OS.

Finally there is still one type of virtualization missing, called OS-level virtualization. This one is the most different approach from the ones we've seen so far. Instead of emulating the hardware, on the OS-level approach the host kernel provides isolated containers that, although can be seen as isolated OS instances, run on top of the same kernel. This type of virtualization is very efficient because applications run directly on the hardware. As a drawback it may present more security concerns, and the guest OS must be the same as the host's one.

There are lots of implementations available, both proprietary and FOSS. On the first group we can find Microsoft Hyper-V and VMware vSphere. On the second we can find Xen (a bare-metal hypervisor) and KVM (an hosted solution).

After presenting the different virtualization approaches it's useful to understand how these technologies can be useful and how are they accepted in the market.

Virtual Machines are enablers to a multitude of applications, from simply eliminating server sprawl to advanced cloud computing scenarios. The decoupling offered by VMs is a key requirement for scenarios requiring very high dynamics. Creating a server is now as easy as instantiate a simple process and comes with a couple of bonus features like live migrations, that lets servers move across the datacenter without downtime, or server resizing, which lets server resources be easily provisioned as necessary. These kind of characteristics are extremely welcome to the highly dynamic requirements we see on the market this days, and that's why this technology is being so widely accepted and so much money is being invested on it.

Network virtualization

Server virtualization has been evolving for a long time, providing by now extremely powerful and flexible solutions for deploying servers. Network virtualization brings to networks the flexibility server virtualization brings to servers, by decoupling the network architecture from the network hardware the same way hypervisors decouple the operative system execution from the underlying hardware.

Network virtualization encompasses a set of technologies and mechanisms that allow the creation of multiple full virtual Layer 2 (L2) – Layer 7 (L7) stacks running on top of standard network equipment. Associated with virtual servers we can provide full virtualized environments, comprehending not only the OSs and applications, but also the network that

connects them, including advanced services like firewall, load balancing, network address translation (NAT) , etc.

Despite of being already in production, network virtualization is a new technology, and more than that, a new concept, and the ideas around it are still being properly defined. We can argue that the simple usage of Virtual LANs (VLANs) is in fact network virtualization, and there's also some confusion around network virtualization being the simple creation of overlay networks, which is not true. But enterprises and the community in general are pushing it forward. What is network virtualization, how the network itself is virtualized and how do we virtualize network functions is a really big research area.

In spite of the possible vagueness of the definition, we can find some common points looking through the literature. Usually network virtualization starts at the hypervisor level, or to be more specific, at the Virtual Switchs (vSwitchs) providing connectivity to the VMs. Packets originated from the VMs are encapsulated using some technology, from the old VLANs to newer technologies like Virtual Extensible LAN (VXLAN) , and are then sent through the substrate network to the target machines: servers running VMs belonging to the same virtual network. This way the physical network acts only as a packet forwarding substrate that interconnects compute nodes.

The virtualization of network functions like, firewalls, is another important aspect of network virtualization. We can refer to at least two different approaches: the first one is to partition physical appliances the same way a physical server is partitioned to run multiple operative systems, the second is to virtualize network functions by creating virtual machines running specific software that implements the necessary functionality.

2.5 CLOUD FRAMEWORKS

With the growing interest that business has in cloud computing it's hardly surprising that we have a growing ecosystem of cloud solutions, both proprietary and FOSS. As always, each solution has its own strengths and weaknesses, that then reflects on its acceptance by the business stakeholders and the community in general. Understanding which are the most promising solutions and how is its acceptance in the market is very useful to make decisions on which platform one should invest. Obviously we can find solutions at any level of the cloud stack, but on the context of this work we are mainly interested in the IaaS side. Following we present the most important IaaS frameworks available nowadays, and also relate them with the cloud computing providers available in the market.

On the FOSS world we can mention five important initiatives: OpenStack [14], CloudStack [15], Eucalyptus [16], OpenNebula [17] and Nimbus [18]. These solutions were already tested and compared in multiple publications [19, 20, 21]. However these publications need to be read with care, since some of the information discussed are already outdated, given the incredible pace at which these solutions are developed and the cloud environment evolves.

OpenStack. The OpenStack project born from NASA and Rackspace as a project to enable massively scalable infrastructures. It is now under active development by a community of a few thousands of contributors backed by more than 70 companies under the supervision of the OpenStack Foundation. OpenStack is in fact a macro project composed by a set of small projects each dedicated to a component of the cloud solution like compute, network or storage. This is the base of multiple IaaS solutions of big companies like Rackspace, Dreamhost, HP or IBM and is also being used to run the business of companies like PayPal or Yahoo among many others.

CloudStack. Initially developed by Cloud.com and Citrix, this solution was then offered to the Apache Software Foundation and is now published with the Apache 2.0 license. CloudStack implements Amazon EC2 and S3 and vCloud APIs and has also its own API, which is a common practice among open source solutions to boost its acceptance on the market, since it allows a seamless migration from other proprietary options.

Eucalyptus. This solution was created on the academic world, being the result of a research project on the University of California. According to [19] it has a limited scalability and some of its modules are closed source what makes it being abandoned in favor of other projects although it is the base of the Amazon EC2 services.

OpenNebula. Is the European solution for cloud computing. This solution also born as a research project and is now being managed by a company founded to support the project. OpenNebula is a little more limited in terms of functionality and community support than other solutions and is mainly being used inside Europe.

Nimbus Project. Nimbus' objective is to provide an IaaS framework for the scientific community, differing from the previous solutions.

Beyond FOSS solutions we can find in the market some other proprietary options like vCloud from VMware or Microsoft Azure from Microsoft. Being proprietary solutions they are not very relevant in terms of the work developed here, and as such they will not be discussed.

2.6 FOG COMPUTING

The term fog computing is not yet very common across the cloud computing community. But it was already used a couple of times before, eventually with different meanings. Here we're looking at fog computing the way it was defined on [1] and also on [22].

Cloud computing is great, but for some applications it lacks some essential characteristics. Decoupling servers from physical machines and its physical location is great for the majority of the applications. But when talking about latency sensitive applications, location awareness,

mobility, geographical distribution or heterogeneity, among others, all of that fails, and that's when fog comes in.

Fog computing wants to bring some characteristics of cloud computing outside the datacenter. The objective is to solve the aforementioned problems but keep the essential characteristics of cloud like multi-tenancy, flexibility, accountability, etc. As defined by Bonomi et al. "fog is a cloud closer to the ground" bringing this idea of cloud computing outside its natural environment.

Chapter Three

Fog computing at Aveiro University

Aveiro University is a very competitive pole of innovation. More than 30 departments run theoretical and applied research projects either internal, in the context of European projects or in collaboration with other universities, research institutes and the industry. The high research activity and the increasing reliance on IT puts a lot of stress on the communications infrastructure and IT services, that must evolve everyday to cope with the ever-changing demands.

This work born from the necessity to evolve the current university's infrastructure and operational models to a more flexible solution that is able to scale according to the requirements, provide innovative scenarios, simplify services management and reduce costs. In this chapter we describe the envisioned use cases, introduce the efforts currently being implemented and ultimately contextualize this work and its goals.

The vision for the future of the university's services covers virtually every aspect of the infrastructure and operational model, ultimately translating the idea of a fully virtualized campus provided to the user under a cloud like model. The developed work focus on the virtualization of the network infrastructure, and so does this chapter. However through the text we try to give some pointers to how the developed ideas could be used as a base for further work, and be integrated within the envisioned scenario.

3.1 MOTTO AND USE CASES

University's IT infrastructure is a crucial component of the institution dynamics. It must be able to support:

- Administrative services (academic, financial, social support, etc.);
- Classes from different departments (with all the diversity of requirements it represents);
- Research projects, that usually represent even more complex technical requirements, and may require infrastructure for cooperation with external entities;

- Infrastructure for demonstrations and conferences;
- Students associations and nuclei activities;
- Generic services like internet access, email, homing, newsletters, surveys, university's websites, etc.;
- Many other services and scenarios according to users requirements.

It isn't hard to see how heterogeneous the requirements are and therefore the services that support them. Given the nature of the institution and the research activity undertaken it isn't expected this diversity is reduced but quite contrary we can foreseen a growth while novel ideas are being explored.

A single infrastructure spreads all across the campus, plus two off-site locations representing more than 4500 university's PCs, plus thousands of intermittently connected laptops and smartphones belonging to students and staff. There are about 200 servers, both virtual and physical, divided across the two datacenters present in the Campus. These servers are responsible to support all the service portfolio and research activities. An optical network provides connectivity to all the campus departments. The off-site poles directly access the infrastructure through a couple of dedicated wireless links. All the infrastructure is currently managed by the central IT services called sTIC.

Giving some details about the current network operations might be useful to understand how the infrastructure is deployed, what are the problems we face and what are our goals. All the departments are interconnected by an optical network, having traffic across departments routed through it. Inside the departments there is a mix of 10Base-T plus 100Base-T technologies providing connectivity to both students and staff, plus eduroam wireless APs. Everything is switched and the isolation of traffic among the logical networks is done recurring to VLANs. So resuming, we basically have switching plus VLANs inside the departments and routing outside them. In the datacenters, however, there is a mix of technologies configured according to the requirements.

The problem with the current state of affairs is the inflexibility of the infrastructure. Deploy a server for a single class or demonstration needs to be prepared with weeks in advance, because all the setup, from the servers to an eventual reconfiguration of some network links, needs to be done manually by the sTIC staff. It gets even worse when we're talking about research. The creation of testbeds usually occurs in a separated infrastructure since it's the only way to have full control of the deployment without threaten the security and stability of university services. The services are also usually coupled to a physical location, what means that change a class to another classroom is next to impossible if we depended on the infrastructure.

We are basically tied to how things gone in the last century. But technology has evolved, tools are better today, we have more resources and mainly more know-how. So its legitimate to expect better services to cope with todays needs. However, the lack of automated processes and the difficulty to manage and evolve an ossified infrastructure leads to a system that isn't

able to respond to the users' needs and consequently leads to an huge duplication of work and resources, ultimately reducing the amount of time and money dedicated to the effective work in favor of routine tasks that are, unfortunately, necessary.

The "cloud computing revolution" we are assisting today is a great opportunity to evolve the current service portfolio. Among the characteristics already presented before we should emphasize three very important ones: (1) it is highly automated, (2) it supports multi-tenancy (i.e. isolation, in what concerns our scenario), and (3) it gives the tenant power to actually control its deployment reducing routine tasks performed by sTIC, and consequently enabling the IT staff to focus on evolving the infrastructure and provide a better service. If we consider the existence of very good FOSS cloud computing solutions that run on top of standard hardware, we can see that a private cloud deployment would probably be a very good way to provide some of the internal university services, without the necessity of a great capital investment but with big advantages to end user, the staff and the university itself.

So a move to the cloud is one of the building blocks to solve our problem, and as we're going to see is also the basis of this work. But cloud computing doesn't solve all the problems, mainly considering that cloud computing is currently something of the datacenter. Cloud computing doesn't move outside the boundaries of the server rooms to the outside campus, and that's where this work comes in. We aim to reproduce the flexibility of a cloud scenario on the physical network we currently own at the university, by automate network processes and make them more flexible, implementing what we previously called fog computing, therefore enabling the provision of innovative scenarios from the simple partition and isolation of network segments, to the mobility of services across the campus and even outside our premises, relying on a mix of physical and virtual environment, supported on an highly automated platform.

Our main goal is to create a network orchestrator tool able to extend datacenter's virtual networking outside the datacenter by virtualizing the physical substrate through the automatic configuration of network devices. The idea is to be able to work with the physical network segments the same way we work with virtual ones, i.e. create virtual networks composed not only by virtual ports and the correspondent VMs but also by physical network segments, at the maximum possible granularity. From the point of view of the user devices connected to this hybrid networks it would appear to be connected on the same L2 segment although they might be geographically distant. We can think about it like the creation of Virtual Private Networks (VPNs) connecting the virtual and the physical environments, however with an important difference: the objective here is to support the technologies available on the network and not the other way around (being the network responsible to implement the required technologies). Nothing prevents the framework to use a VPN technology though, but that might not be possible due to devices constraints, policies, performance or others, so the it must be possible to use whatever technology is available on the network to perform the job (VLANs for instance), or even multiple technologies at once. That's why we're calling it orchestrator, it should be able to orchestrate multiple technologies and/or devices to achieve

its goal. A very important aspect is that the framework cannot affect already running services, so for example using VLANs, the orchestrator cannot use an ID that is already in use to isolate some logical network.

Following we describe a set of exemplification scenarios that illustrate some of the use cases we want to deploy as part of the university's services. The objective is not to provide a technical description of the project requirements but rather give the reader some examples of the ideas behind the project and the expected outcome. Later, on chapter 4, we properly describe the technical requirements that are implicit in the scenarios presented bellow.

Figure 3.1 represents a simplified network infrastructure used to describe the scenarios. It is obviously extremely simplified considering the real university network but is complex enough to let the reader understand the ideas. As you will notice reading the through the scenarios, most of the ideas are already technologically feasible, what should be emphasized here is how the services are to be provided: everything is dynamic and provided to the user on demand with all configurations applied to the network equipment automatically without technical staff intervention. A big change from what we currently have.

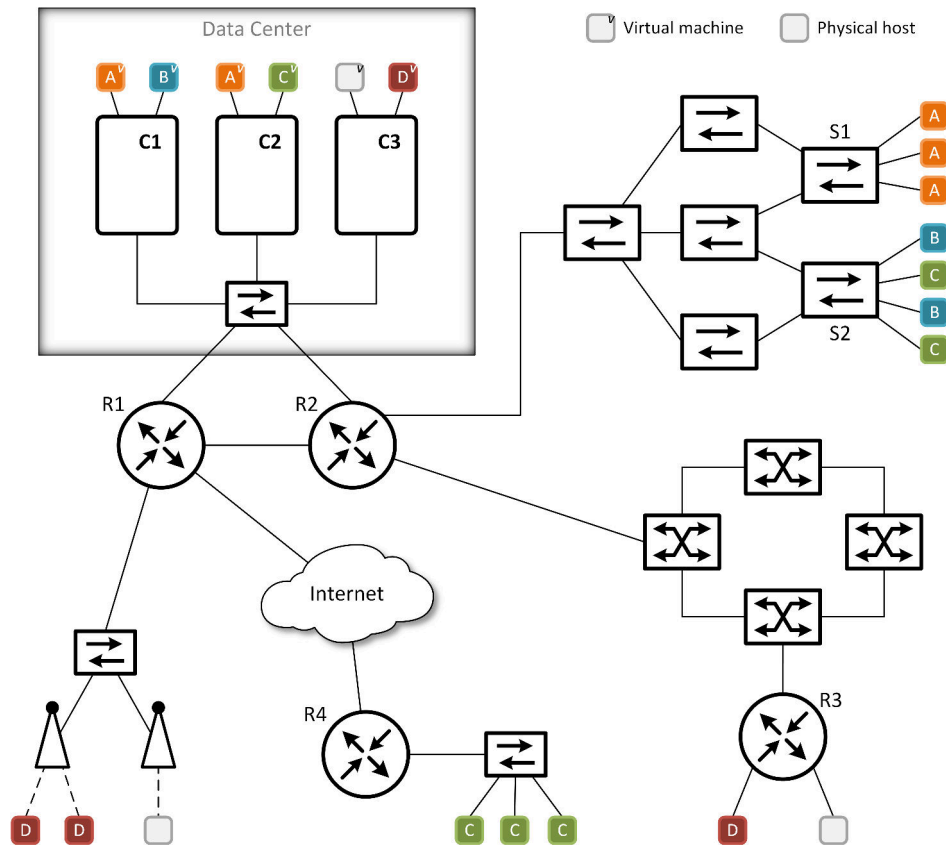


Figure 3.1: Simplified physical network infrastructure backing the exemplification scenarios.

3.1.1 Scenario 1

We currently hold some classes that depend on the physical servers present in the classroom, namely networking related classes. However servers are getting old and replacing them is expensive, so we want to virtualize and move them to the datacenter but keep everything working as if they are directly connected to the classroom network. Since most of the experiments involve Dynamic Host Configuration Protocol (DHCP) servers and sniffing L2 packets, all the network traffic must be isolated from the datacenter to the room where the experiences take place.

The first step is to create the virtualized version of the servers and connect them to an isolated virtual network using a cloud computing solution deployed on the datacenter. After deploying the servers the user simply has to specify that the virtual network the servers are connected to should be connected to the specific room where the classes take place (selecting it from a list for instance). Let's say that switch S1 from figure 3.1 is placed on the room and hosts marked with A represent our servers and students computers. From the point of view of the hosts the network should look like figure 3.2, although servers and student's computers are physically distant.

The user deploying the scenario should be able to define the network topology through a simple interface, ideally the same interface used to deploy the virtual servers. The configuration of the physical network, both in and outside the datacenter is executed automatically as soon as the user specifies the topology.

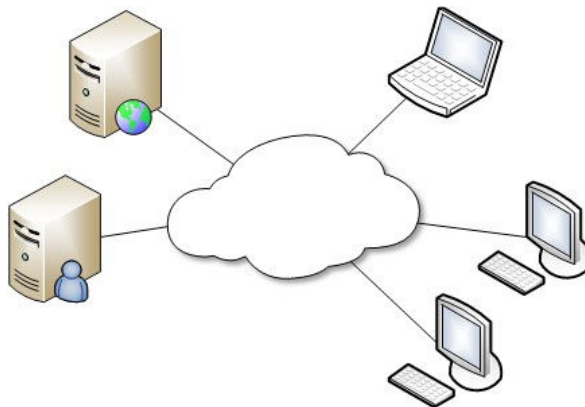


Figure 3.2: Virtual network described on scenario 1 deployed on top of the physical network sketched on figure 3.1. The PCs and the laptop represent the hosts connected to switch S1 and the servers represent the virtual machines marked with A.

3.1.2 Scenario 2

Consider the first scenario. We now want to use the virtualized servers to teach the same class on another distant classroom. Using the same interface as before the user is able to request that the servers are now connected to the new classroom instead of the previous one. All the configurations is automatically handled by the system, as previously happened, and applied as soon as the user requests the modification.

The same should happen if instead of changing the classroom we want to change the connected servers to teach another lesson, or want to connect multiple classrooms at the same time. The user can simply change the topology of the deployment and specify which physical segments are connected to a virtual network.

3.1.3 Scenario 3

We're now conducting experiments in which we need different groups of people to access different servers. Consider that some of our users are connected to the managed switch S2 on figure 3.1. Through the same interface as before, we are able to specify to which virtual network each one of the switch ports is connected to, letting different users be connected to different virtual environments. But it isn't limited to the same switch our room. We can even have users connected over the internet, creating what we can see in the figure represented by the hosts B and C forming two different virtual networks.

Our objective is to be able to configure the physical network with the maximum possible granularity. How far can we go basically only depends on the devices we have on our network. Using a non-managed switch would probably invalidate this use case.

3.1.4 Scenario 4

We want to make a demonstration of a new service developed in the university. The development was carried using a virtual environment on the datacenter, and for the demonstration we want the assistance laptops to be directly connected to our testing environment, avoiding have to request sTIC for public IP addresses and access to the university's network for foreign people.

We can request our service the configuration of a new Service Set Identifier (SSID) on the wireless APs available on the demonstration room, that provide a direct connection to our virtualized environment. Users connected to new SSID have access to our service, but only to our service, easing the demonstration setup work and reducing possible security risks resulting from having foreign people connected to the institutional network.

This scenario corresponds to the hosts marked with D on figure 3.1.

3.2 CLOUD FRAMEWORK

We said before that cloud computing is one of the pillars of this work. Hopefully with the scenarios presented before we can see the dependency on a cloud infrastructure. Therefore this work comprises the integration of our new framework with the environment already provided by cloud computing solutions.

The chosen platform was OpenStack. Analyzing the available offers we can give multiple reasons that lead to that decision, the most important are:

- It's a FOSS solution, what tremendously eases the development of extensions;
- Its architecture, divided in multiple modules/projects, is by nature extensible;

- Has a great community;
- It's currently the most used FOSS cloud framework;
- There are dozens of companies currently investing in OpenStack;

We should add to the previous list the fact that OpenStack is already being tested and even used in production inside the university, as we will explain next.

3.2.1 Development testbed

This work was developed on the premises of Advanced Telecommunications and Networking Group (ATNoG) , a research group from the Institute of Telecommunications. In order to experiment with cloud computing we currently own two deployments of the OpenStack cloud framework. The first one, intended for production, is currently supporting the ATNoG services portfolio including multiple websites, mail server, Domain Name System (DNS) server, project management software, among many other. The second one was primarily used as a testbed for OpenStack experimenting and development, and is currently being migrated to be used in production, supporting the research group activities.

The production environment is composed by a blade system with 12 cores and 200 GB of memory, supported on an 2TB Network File System (NFS) shared storage (provided by sTIC), plus two more nodes running the OpenStack controller and the OpenStack networking components. This deployment is running the Folsom release of the OpenStack framework and is currently supporting more than 30 virtual machines under active use.

The testbed counts with 8 computing cores and 28 GB of RAM distributed across 4 computing nodes with 2TB of NFS shared storage, plus 3 more nodes, one for the controller, one for the networking and the last one for the shared storage. It is running the Grizzly OpenStack release.

The testbed was completely deployed and maintained in the context of this work, providing a very good learning and development tool. All the tests and development were first performed on the physical infrastructure, and later, with a stable deployment of the framework, creating virtualized environments using the full potential of the OpenStack solution.

3.3 OPENSTACK PLATFORM AND ECOSYSTEM

Since we want to integrate our work with the OpenStack platform it's useful to understand what is in fact OpenStack and how that integration can be achieved, both from a technical perspective (what is the software architecture, what facilities are provided for extension, what technologies are being used, etc.), but also from an administrative perspective (what's the release cycle, how does changes get accepted in the framework, how is the project managed and by whom, etc.). That's what we are going to present in this last section of this chapter.

First of all it's important to know what is OpenStack. From the official website, OpenStack is a framework for "building massively scalable clouds running on top of standard hardware" [14], which is great for marketing but doesn't really say a lot. OpenStack is a

macro project comprising, at the time of writing, seven core components, each one geared to solve a different aspect of the cloud computing problem. Together these components are able to orchestrate pools of compute, storage and networking resources to build virtualized infrastructures on demand, everything controlled from a central web dashboard.

3.3.1 OpenStack ecosystem and governance

OpenStack is a global collaboration of software developers with the common objective of building the *de facto* FOSS cloud computing platform. According to the official website [14], there are currently more than 70 companies sponsoring the project, and more than 7000 people from more 850 organizations are already members of the OpenStack Foundation. The funding has already surpassed \$10 Million dollars. Numbers that clearly shows OpenStack's dimension.

The project was started by Rackspace and NASA back in 2010 but is now managed by the OpenStack Foundation, a non profit organization born to promote the development and adoption of the OpenStack platform. Its higher entity is the Board of Directors whose members come from the sponsoring companies. The Board of Directors is responsible for the strategic and financial guidance to the project and community. Helping the Board of Directors there is the Technical Committee (TC) , leading the technical related aspects of the project, and the User Committee helping with the relationship with the OpenStack users.

Ultimately, every major decision related with the course of the project must be approved by the Board of Directors. But the technical guidance inside each component is given by what's called the core developers team. A team of developers that is responsible to manage the project at a technical level, mainly doing code review and decide about proposed changes. This team is headed by the Project Team Leader (PTL) , the core team member that is part of the TC.

Up to the TC, we find on OpenStack a meritocratic community. The members for the core team are proposed from time to time based on their contribution to the community, not only based on submitted code, but also on code review, technical guidance, and the overall participation in the project.

Changes that doesn't greatly affect the course of a project are usually accepted by the core team, and ultimately by the PTL of the project. The process to propose a change starts with the writing of a Blueprint. The BP is the description of what we propose to do, that should clearly describe the proposed changes, how they are going to be implemented and how they affect the component and the interaction with other components. The BP is then analyzed by the community and the core team eventually accepts or rejects it.

To ensure the quality of the code integrating the OpenStack project every patch submitted is firstly tested and peer-reviewed, and only then, if accepted, merged into the master branch. If the code isn't accepted it may either be revised by the author until problems are fixed, or simply dropped if it is decided it doesn't makes sense to integrate it.

The development infrastructure is based on a mix of technologies and services, both

public and deployed on proprietary infrastructure. For project management the project uses Launchpad, Github is used as mirror for the projects' Git repositories. For the remaining services (e.g. the actual Git repositories, code testing, mailing lists and websites, etc.) a private infrastructure is used.

Finally another important point is the release schedule. OpenStack currently holds a six month release schedule, due in October and April every year. The changes to hit a certain release should be proposed up to about 9 weeks before the release, and new features should be merged into the master branch up to about 6 weeks before the release.

3.3.2 OpenStack architecture

OpenStack is greatly based on the SOA paradigm already introduced before. The cloud framework is divided in multiple components, each one providing a well defined service that implements part of the cloud computing environment, and accessible through a proper API. Currently OpenStack counts with 7 core components that we describe next. On this section we only provide an high level overview of each of the components and how they interact with each other, not about the internals of each component. The networking component, the most relevant one in the context of this work, is then detailed on the next section.

- **Nova (Computing):** The Nova component is responsible to manage the VMs through all their life cycle. This includes all the necessary interactions with other components, to attach network interfaces, block storage, etc. Migration of VMs is also managed by this component, so as any other functionality related with VMs themselves. Schedule of VMs across compute nodes and the overall management of the nodes in what relates to compute resources availability is also managed by nova.
- **Neutron (Networking):** Neutron, previously called Quantum, is the networking component of OpenStack, therefore responsible for all the network related operations. It's responsible to setup vSwitchs, launch DNS servers, configure routing tables, among other tasks that permit the deployment of virtual machines inside isolated virtual networks. Neutron is also capable of interact with external physical devices in order to setup the physical network inside the datacenter and to handle VMs' traffic. This is the most important component of OpenStack concerning the integration of this work.
- **Cinder (Block storage):** The Cinder component is responsible to deploy virtual block storage devices, that will then be available as block devices to the VMs. It can use a multitude of backends to effectively implement the block devices, like Logical Volume Manager (LVM) volumes, or simple files.
- **Swift (Object storage):** Swift implements a storage model alternative to the standard block devices, providing an API to store and retrieve individual objects (like generic files, disk images, or any other).

- Glance (Image): This component provides an efficient and easy to use model for storing disk images, that can be used to boot VMs. It can be based on Swift for object storage, or can simply rely on local file storage.
- Keystone (Identity): Keystone handles all the authentication and authorization. This includes the management of users and tenants, but also the authentication of the different framework components. Given its nature, every OpenStack component interacts with Keystone.
- Horizon (Dashboard): Finally Horizon is the dashboard through which we can interact with the OpenStack framework. It is web based and implements the majority of the functions provided by the OpenStack components, although sometimes the projects are not totally synchronized. It uses the APIs to interact with all the other components.

A very important aspect we should emphasize is the reliance on plugins. If we look through the aforementioned services, we can see that all of them depend on a backend technology to execute its jobs. It can be the hypervisor (for Nova), a virtual switch (for Neutron), an object storage facility (for Glance), Keystone depends on a token storing backend, all of the components (except for Horizon) depend on a database, etc. To make the framework generic, and not tied to a specific backend solution, all of the components have the notion of plugins that enable the usage of other backends. For instance, in the case of Nova, there is a plugin for the KVM hypervisor, another for Xen, etc. This is a very important structural aspect since it extremely increases the extensibility of the framework.

3.3.3 Neutron: the network service

Given the nature of this work, Neutron is probably the most interesting OpenStack component for us. Lets start by looking at what can it already do and then at how can it be extended.

Neutron is the network service component of OpenStack. What this means is that it is responsible for deploying all the network related services, that includes per-tenant virtual networks, properly isolated, with its own IP addressing space (that can eventually overlap with other networks), routing, NAT and firewall. Currently it is also possible to insert external services, like physical appliances, in the virtual network, through what is called service insertion. Neutron is able to manage not only virtual switches, like OpenVSwitch, but also physical devices, mostly OpenFlow controllers. Considering the implementation of the virtual networks, Neutron is able to use any isolation technology through the use of plugins. Currently VLAN, VXLAN and Generic Routing Encapsulation (GRE) are available out-of-the-box.

All the Neutron architecture is based on plugins. Neutron server is basically responsible for implementing an API and manage generic database data. The actual deployment of the network functionalities is done through plugins. The base plugin to deploy Neutron is the core plugin. This is the plugin that defines the segmentation technology and the vSwitch to use and effectively deploy the service by configuring the virtual devices.

One of the most common plugins is the OVS plugin (OVS standing for OpenVSwitch). As the name implies, this plugin uses the OpenVSwitch as backend. It is able to use VLAN, GRE and also VXLAN as segmentation technology. Routing is done through the kernel facilities (on linux net namespaces are used to provide isolation). For DNS, dnsmasq is commonly used but it's possible to use a different software. For the firewall iptables are used under linux.

Neutron architecture comprises basically two major components. The Neutron server properly extended by plugins (already introduced), and the agents. The agents are dependent on the chosen plugin and it's through them that neutron actually deploys the required services, i.e. actually implement the configurations. The communication among them is usually implemented through message queues, like rabbitmq or zeromq (similarly to what happens in almost all of the other components).

Neutron was designed to support only one core plugin at a time, meaning that a deployment is limited to the use of only one technology. In order to support more advanced scenarios, during the Havana development cycle a new plugin called Modular Layer 2 (ML2) was developed that enables multiple technologies to be used simultaneously. It even allows the same virtual network to be composed by multiple segments based on different technologies. This plugin is important because it changes the way we extend Neutron.

Originally, extending Neutron is done through the development of plugins that implement support for the intended backend. With the appearing of ML2 it is now possible to support new backends without all the burden associated with the development of a new plugin, that usually represents managing plugin related state on the database, among other aspects that lead to code duplication.

ML2 introduced two new important concepts: (1) type driver and (2) mechanism driver. The type driver is responsible for managing backend related information (e.g. VLAN ids when using VLANs for isolation). The mechanism driver is responsible for interacting with external devices when that is required (e.g. when using an external OpenFlow controller).

Chapter Four

System Design

On the previous chapter we contextualized this work and provided an overview of the problem and the main goals described from the user perspective. We have now a set of use cases and functional requirements that define what we're proposing to achieve and it's time to move from the problem to the solution.

To develop a successful solution it is essential to clearly describe our ideas from a technical perspective, discuss the challenges we're going to face and finally, take the major decisions that will drive us through the solution design process.

On this chapter we start by technically describe our goals, enumerate the requirements and identify possible problems, and based on that, discuss some major decisions. Then we move to the proposed solution and describe it conceptually in the first place, later moving to implementation related aspects.

4.1 REQUIREMENTS AND MAJOR DECISIONS

Technically speaking we want to build a network virtualization framework since, in simple terms, the final objective is to deploy virtual networks on top of our physical one. But network virtualization is a broad term, and there are already multiple tools claiming to implement it (including OpenStack), so we have to properly define what we mean by network virtualization and what are our specific requirements and constraints, that differentiate this solution from any other.

The first step should be to define which network layers or services we want to virtualize. From the use cases we can see the objective is to create isolated layer 2 domains on top of the physical substrate. So, as a first requirement, we can define that our solution should be able to virtualize L2 networks, being upper layers outside of the scope.

We saw on chapter 2 that L2 network virtualization can be achieved using different technologies. So it's now important to look at what technologies we can or should use to achieve our goal. Looking again to the use cases we can see that, unlike what usually happens with these virtualization frameworks, our solution should be able to support any technology

required by the final user, and more than that, it should be able to orchestrate different technologies to provide a single virtual network. Therefore, it's the framework responsibility to adapt to the network requirements and not the other way around.

On the line of the technologies we can also see that our framework cannot be bound to specific devices or even vendors. The solution should be able to operate any device found on the network. Regarding this field, an important aspect is the integration with the cloud environment. Integrate with the cloud means we have to consider the management of virtual network devices in addition to the physical ones, mainly virtual switches since they are the base for providing network connectivity to VMs.

These two last aspects point us to the necessity of building an extensible solution, since it is impossible to support every device and technology currently available and appearing in the future.

Moving on into the requirements, we see that being able to work on top of a production deployment without disturbing already running services is another very important one. What this means is that our framework may not have the full control of the network devices and the network itself, so operations like rebooting a device, or cleanup configurations are completely out of the question. This aspect will distinguish it from an automated network management tool and make it closer to just another service running on the infrastructure.

Another desirable requirement is that the framework can respond to changes in the network environment. There are two scenarios here: (1) the change is planned by the user and properly registered on the framework, e.g. the user is adding a new router to the network, and (2) there are unplanned changes in the environment, e.g. a hardware failure occurs. The first scenario doesn't bring any concern beyond having in mind that the network is dynamic, but the second scenario can place a big issue: device monitoring. If we consider that every time we want to check the state of a device we may need to initiate a telnet or an Secure Shell (SSH) session isn't hard to see that as the number of devices grows, the device management process may become an obstacle to the scalability of the solution. So this is a very important aspect to consider when designing the device management aspects of our framework.

Integrate with a cloud computing solution is one of the central points of the work, therefore we need to carefully analyze the implications of such integration both from a conceptual and from an implementation perspective. The major point to define, as that will change the way we look to the problem, is whether the solution will be developed as an extension of the cloud framework or will be a standalone solution. The chosen platform is OpenStack, which is by nature easily extensible. Thus, develop an extension is probably the easier and quicker solution since OpenStack has already a lot of useful capabilities like multi tenancy support or virtual network management and a good code base to develop the necessary APIs, manage Databases (DBs), create the Command Line Interface (CLI), etc. But developing for OpenStack has multiple drawbacks: it confines the usage of this new framework to a cloud environment (although it would probably be useful as a standalone solution for network management); the solution design will be driven by how OpenStack works, possibly leading to

suboptimal decisions to ease the integration; porting the solution to any other cloud platform may become very difficult or even impossible. On the other hand, develop a completely standalone solution will require a much larger effort since all that support OpenStack would offer will need to be created from scratch, eventually leading to duplication of data (and therefore possible inconsistencies), and would probably deviate the project from its main objective, solving already solved problems.

So the solution will be somewhere in the middle. The proposed framework will be a standalone solution but the responsibilities will be split among the two frameworks. Our framework will basically be responsible to manage the physical network topology and deploy virtual networks on it. No management of users and tenants, quotas, authorization, Graphical User Interfaces (GUIs), etc. OpenStack will be responsible for the interaction with the end user and instruct the framework about the virtual networks to be deployed, and will only be extended with the minimum essential structures. We can say that from the point of view of the OpenStack our framework will be an external network controller, and from the point of view of our framework OpenStack will be a manager. The concept of manager will be explained on section 4.2.4, but it basically represents the external components that interact with the framework. It can be OpenStack, a monitoring tool, a person using a CLI, etc.

To make it generic and easily portable all the interactions with our framework will be performed through properly defined interfaces. This way it would be easy to port the solution to other cloud framework, or implement standalone tools to interact with it. We can easily see two different interfaces here, one for the management of the physical network, and another for the management of the virtual ones. Interactions with the end users will be provided by OpenStack, that will then translate the virtual network requests to the proper framework interface. Although it would be possible to also manage the physical network using OpenStack, it would pollute OpenStack interfaces with operations most of the time only available to the cloud operator. As so, the physical network management interface should be implemented by a standalone client.

This approach joins the best of the two worlds providing a generic lightweight and easily portable network orchestration tool. The OpenStack framework will also need to be extended by implementing support for the new concepts that will be introduced and to interact with the API provided by this new framework, but won't manage any state related to the physical substrate.

All the details and specifications will now be described on the next sections, starting by the creation of the conceptual abstractions and then moving to the implementation details that include the necessary changes to OpenStack.

4.2 CONCEPTUAL SOLUTION AND MACRO BLOCKS

Following the ideas discussed in the text above we will now present the reader with the abstractions we created and the macro blocks that will comprise our solution. Although we

do not intend to give implementation details here, we will give some information to ease the correlation with the implementation section 4.3.

4.2.1 Network abstraction

The physical network can be extremely complex. Composed by different devices like routers, switches, APs, firewalls, load-balancers, from multiple vendors, with different capacities, different management interfaces and a lot of other different characteristics, interconnected using some specific architecture eventually in a sub-optimal way patched according to the available resources at a time. But most of this complexity isn't relevant to our system. Using a vSwitch, a physical switch or a router to tag packets with a VLAN id isn't important as long as the packets are properly encapsulated, the number of hops an Layer 3 (L3) tunnel have to cross, how the devices are configured, and many other details can be abstracted.

A network abstraction was therefore designed with the intent of simplifying all the process of creation and management of the virtual networks. The focus was put on extract only the relevant information from the physical network, simplifying the process of creating virtual domains and map them to the physical substrate.

The abstraction is composed by four concepts. Node, port, segment and link are the four entities that should be able to describe all the physical network topology and all the necessary configurations our system wants to apply on the devices. Below we describe each one of the four entities, their responsibilities and how they interact with each other.

Node. A node represents a device, any network device that can be configured by our system, including virtual devices like virtual switches. To make it clear, a node doesn't necessarily map one to one with physical equipment, mainly in what relates to virtual devices, since multiple virtual devices, and therefore multiple nodes, can run on the same hardware. A node represents a large number of different entities, the same concept can map to devices so disparate as routers, APs or a simple linux box because they share three main responsibilities: provide ports, create links and connect links to ports or to other links. Nodes are connected to each other through what we called segments.

Port. A port is the point of access to the virtual network. User devices, that can range from VMs to smartphones or anything else that can generate and/or receive packets, connects to the virtual network through them. Ports, in the same way as devices, may therefore represent lots of different realizations. Some examples are a switch port, an WiFi network or a virtual port on a vSwitch. There is a very important requirement though: the traffic that flows through a port must be isolated, i.e. the node providing the port must support some technology that can isolate the traffic from the different ports, for example VLANs. A good example is an AP that can send the traffic of each of the provided SSIDs to a specific VLAN on the wired network, in which case every provided SSID can be considered a port. On the other hand a non-managed switch cannot provide any port because traffic cannot be isolated.

Segment. A segment represents a domain for the creation of virtual links. Each segment is basically associated with a technology and owns a pool of identifiers. Nodes connected to a segment can create virtual links among them, and is the responsibility of the link entity to describe how each virtual link should be created, i.e. among which nodes and using which identifier from the segment pool. Valid examples of technologies are VLAN, GRE, VXLAN, etc. The segments can map to a network segment or not, for instance a segment using GRE to provide isolation can represent all the Internet, but a segment using VLAN will probably map to a physical segment of the internal network that carries 802.1Q packets.

Link. A link is a representation of the connections among nodes in a segment. Each link describes the segment it refers to, what nodes on that segment are connected, and the identifier they are using for that link. Traffic from a virtual network crosses each segment isolated according to the link associated with that virtual network on each segment.

So, to summarize the concepts introduced before, a network is composed by nodes and segments. Nodes provide ports and interconnect segments. Ports are the connection point to the network and links describe the virtual links created on top of a segment. With this information we can fully describe all the environment, both the physical substrate and the virtual deployment.

4.2.2 Virtual network mapping

Using the abstraction proposed before, a virtual network can be basically described as a set of ports. Deploying the virtual network consists on connecting the nodes providing those ports by creating virtual links on each of the necessary segments. So, a list of ports and a list of links completely describes a virtual network and its deployment on the physical substrate.

We reach now the problem of mapping the virtual network on top of the physical network topology. The objective is to get a list of links that connect all the nodes providing ports for the virtual network, as explained above. There is a very important point to consider here: as we're talking about a L2 network we must ensure there are no loops on the virtual network.

The first logical step to solve the mapping problem is to find a suitable solution to represent the physical network. We are basically representing the connectivity among a set of points, so, an evident approach is to use a graph composed by nodes (the vertices) and segments (the edges). Ports and links do not represent the physical topology, so they don't need to be considered here. Using a graph, the virtual network mapping is reduced to a problem of finding a tree that connects the necessary vertices (nodes) on the graph.

As the physical topology complexity grows it's probable that we can find multiple solutions to the map the same network. So, beyond finding a solution we now have the problem of what solution to choose. To pick one we probably need to define what is the best solution, which is not a trivial problem since there is no such thing as the best solution. It depends on the user requirements, which can be maximize throughput, maximize link utilization, prefer a set of segments, distribute the links geographically, or any other more or less exotic idea.

Although it is an important part of the solution, this work is not about virtual network mapping. So, the focus here was not on the algorithms to be used but more on the framework support. We can see two points: (1) the framework must be able to store and retrieve random information about the network topology, because the user can define what information should be stored about the network topology, and (2) the mapping algorithm should be pluggable to give the user all the freedom to use whatever fits it better to him.

To implement the first point, all the four entities defined on the abstraction will be extended with a list of key value entries, that let both the user and the mapping algorithms store any necessary information. We can imagine for instance priorities on the segments, packet statistics on the links and ports, geographical position on nodes, etc. As the user is able to read and write this information any external entities, like a monitoring tool will be also able to update this information in real time, letting the mapping algorithms perform its job the better possible way.

The second point is a simple matter of providing a proper interface for the mapping algorithms, plus dynamically loading the mapping algorithm according to the configurations provided by the end user. We named the interface `vNetMapper`.

4.2.3 Device management

The management of the network devices is the final purpose of this framework. Therefore it isn't surprising that this is a very important part of the solution. As we discussed before, due to the necessity of monitoring the devices' state, we should segregate the device management stage from our main network controller, and as so, we created the CNa. As a reference, the main network controller is called CNc and will be introduced on the next section. For now it's only important to know it exists.

CNa is then the framework component responsible for the management of the devices. This is an independent piece of software, that will run independently of the main network controller, with only two responsibilities: apply the device configurations and report its state. This architectural detail is very similar to what OpenStack already deploys, since OpenStack also relies on the concept of agent to effectively apply the desired configurations on the managed vSwitchs.

Being able to execute the device management on an distributed manner brings a lot of flexibility to the deployment and improves performance, reliability and manageability. But this model may also increase the complexity of the infrastructure, mainly for small deployments. To prevent that, each CNa instance may handle from 1 to all of the network devices. This will give freedom to the system administrator to choose what is best for its deployment. By implementing it as an independent component we also encourage alternative deployments (e.g. integrate the agent on the device being managed), and alternative usages (e.g. integrate the agent in a monitoring framework).

By reading the text above we can clearly see that the CNa component can itself be split in two different parts: (1) device configuration, to effectively deploy the desired configurations

and create the intended network topology, and (2) device monitoring, to manage the device state and trigger the necessary actions accordingly to eventual state changes, namely remap the virtual networks in case of device failure. Each of these responsibilities alludes to an interface. The first interface, called `devConf`, describes the communication from CNc to CNa and is meant for the configuration of devices. The second, called `devMon`, implements the opposite communication direction and is meant for the CNa to report the device state. One of the major advantages of having these two different interfaces is that the CNc is able to implement a full asynchronous communication model with the CNas.

One of the possibilities to implement the communication among both components is to use a message queue like RabbitMQ or ZeroMQ, using an approach similar to OpenStack that uses Remote Procedure Call (RPC) over these mechanisms. This aspect will remove the burden of developing the communication mechanism by providing reliable communication channels. Sharing the queue server with OpenStack is also a possibility, simplifying the deployment of the framework.

It's now time to look at the effective management of the nodes. The concept of node, presented before, abstracts the functionality of a huge number of different devices, from different vendors, with different management interfaces, different capabilities, using different technologies, and many other differences. A node can represent so disparate devices as an AP and a router. To cope with this enormous diversity we use the concept of driver. Each different device must be supported by a driver that translates the node API calls to effective configurations on the physical device.

To be able to execute the configuration of the device, the driver must be able to answer two distinct questions: (1) how to manage the device and (2) how to convert the API operations to device configurations. The first point includes knowing the device management interface (i.e. the commands available, the syntax, etc.), how to access it (e.g. a CLI via telnet or SSH, or even something more complex like Simple Network Management Protocol (SNMP) or a REST interface), the device state, including the physical configuration (what ports are connected where, how many interfaces are available), the device capabilities (what technologies does it support, how many bridges can be created, etc.) and the mappings between the physical and logical configurations. This first point is typically the common part among devices from the same vendor and may be based on existing libraries or give birth to the creation of those. The second point means the driver will have to know what creating ports means, how to deploy a link, how to interconnect links or links to ports, etc. This basically depends on the device functionality and on the implementation, and ultimately will have to be analyzed on a case by case basis. An important aspect here is that the same device model may not always be associated with the same driver, this gives freedom to the network administrator to even code a different driver to cope with very specific device configurations or requirements, what gives a huge flexibility to the framework.

There is a requirement for the device driver that is worth recalling. Independently of how the device is configured it must always be ensured that the Layer 2 information is kept among

segments, since the final goal is to have an L2 virtual network.

Finally, to ensure every device is independent of the others, we will have per device configuration files. These files will have a common part, that specifies the device name, device driver, etc., and a driver specific part, that will provide driver specific information. These files are provided to the CNas upon initialization.

4.2.4 Network controller

There's only one component left to present: the main network controller, which we called CNc. As this is the central component of our solution, some of its responsibilities has already been introduced throughout the previous sections, when we talked about the interactions with the other components. Therefore, on this chapter we'll give a summary about the CNc and describe only the missing parts, that mainly relates with its interaction with the managers.

CNc is the core of our framework. There is only one instance of the CNc per deployment, that is responsible to: (1) interact with the managers, (2) handle the database, (3) execute the mapping algorithms, and finally (4) communicate with the multiple CNas, both to configure the devices and to receive the state reports. The concept of manager is very important here. It represents any external system that interacts with our framework.

Managers may be responsible to inform the CNc about changes in the physical network (e.g. create segments, update nodes, etc.), instruct about virtual network operations (that basically resumes to instruct about which ports belong to each virtual network), or both, what leads to the creation of two different APIs. OpenStack for instance, will only be responsible to handle the management of the virtual networks. Managers can also register for receiving information updates about changes in the deployment, again both for the physical, virtual networks or both of course.

The first interface, called `vNetConf` will provide methods for managers like OpenStack request the configuration of virtual networks. The second one, called `pNetConf` will let managers configure the physical substrate. Both interfaces will let managers retrieve information about the network state, but a different subset of information. For the first case, all virtual network information is provided, plus information about the existent nodes and ports, since this information is necessary to provide an user interface. For the second interface all the information about the physical network is provided plus the created links and associated virtual network.

With these two interfaces all the network operations can be accomplished, but managers will have to poll for changes on the networks to keep updated about its state. Polling might not be the best solution as the network grows, therefore an asynchronous solution is provided using a couple more interfaces called `vNetMon` and `pNetMon`. These interfaces should be implemented by the managers and an handler should be registered on the CNc. This way, the CNc will be responsible to inform the managers about eventual changes in the networks state as they happen.

Summing all up, our framework comprises two big components: CNc and CNa. CNc is

responsible to control all the service, and each instance of the CNa is responsible to manage one or more nodes. CNc will provide four interfaces for bidirectional asynchronous communication with two different types of managers. For the communication with the CNas there are two more interfaces for bidirectional communication.

4.3 IMPLEMENTATION

On the previous sections we introduced the reader to the major concepts behind our framework and gave the rationale behind our decisions. The information presented above is basically what the end user will need to understand to give the best use to our platform.

On this section we will move to implementation related aspects. Although we do not provide an implementation of the framework as part of this work, and therefore some specific details will be missing, we intend to provide all the necessary architectural information to enable a future implementation. To have formal specifications we will resort to the use of Unified Modelling Language (UML) diagrams whenever possible.

Not having an implementation means we're not bound to any specific programming language. However, in the tests executed during the development of this specification, Python was the chosen language. Therefore we can find some details influenced by the libraries and frameworks available for the Python language, namely Python SQLAlchemy [23]. Some details are also inspired on the OpenStack framework and therefore we can find some similarities.

4.3.1 Software architecture

When describing an object oriented architecture, the class diagram is probably one of the most crucial aspects of the design. We have two major components on our framework, and therefore we created two class diagrams describing the architecture of each one of the components. Obviously there will be overlaps and some of the classes and interfaces will be present on both diagrams.

The class diagrams for both of the components are available on appendix A. Figure A.1 presents the CNc class diagram and figure A.2 the CNa one.

Referring to the CNc component the main class will be the **CNc**. This class implements the base logic of the component. It's responsible to instantiate all the other necessary classes and manage the interactions among them. It also defines the entry point for the CNc application, i.e. the equivalent of implementing the main function.

The **CNc** component should implement a set of interfaces that represent the public APIs (that we'll describe next). The **pNetConf** will let managers define and control the physical network topology, so it will provide methods like **createSegment** or **attachNode**. The interface **vNetConf** is similar but is meant to control the virtual network deployment so the methods will be for instance **createVNet** or **requestPort**. Finally the interface **CNcMgr** provides a couple of methods for the registering of managers. It is through this interface that a manager will register itself in the framework and request to receive updates about the physical network, virtual networks or both.

The class **pNetMgr** represents a physical network manager. It will hold information about the manager and will be used by the **CNc** to provide manager with updates. It abstracts the **CNc** from details on how to communicate with managers. This class should implement the **vNetMon** interface, and so do the actual manager.

The class **vNetMgr** is similar to the **pNetMgr** but is intended for virtual networks. The same happens with the **vNetMon** interface.

For the mapping of the virtual networks we can find two classes and one interface. The class **Mapper** is the responsible to implement the mapping of the virtual networks on the physical topology. As defined before this is a plugin, and the **CNc** will be the responsible to instantiate the correct implementation according to a configuration provided on startup. To be a valid plugin, implementations should comply with the **vNetMapper** interface. Finally, the **Mapper** will be provided with a configuration object on instantiation, the **MapperConf**, that will carry the user configurations for the plugin. Beyond the generic properties this class should also provide a key value pair repository to store specific (per plugin) information.

The **CNaMgr** class is the responsible to handle the interaction with the multiple instances of the **CNa** component. To keep track of the registered **CNas** it will use the **CNaInfo** class that will hold information about each **CNa**, including what nodes does it manages (as we'll see next). The **CNaMgr** class implements two interfaces, one for each communication direction. The **devConf** interface will provide the methods for the **CNc** class send commands to the **CNas**, and the **devMon** represents the interface the **CNa** will use to report the device state to the **CNc**.

Now we move to the physical network information. As we'll see when we talk about the database, these classes will be directly used to persist physical network information. So it's natural we have a class for each one of our network entities, namely we have the **Node**, **Segment**, **Port** and **Link** classes that will represent each of our components. There are also some auxiliary classes. A **Link** is a set of associations between nodes in the same segment, so we provide the **Connection** class that represents each one of those associations. Then we have the **Element**, that is an abstract class, and the parent of the each one of the previous four classes, used to provide a set of properties to each one of them. This properties, represented by the **Property** class, are the used to store the extra information for the mapping algorithms as discussed before. To manage all the physical network we have the **PhysNet** class. Finally, the **VirtualNetwork** class is used to associate the entities with the virtual network they belong to, if any.

Moving on to the **CNa** we can start by looking at the main class, the **CNa**. This class is, similarly to the **CNc** one, the base class for the **CNa** component, that will implement the main logic for it.

The purpose of the **CNa** is to manage nodes. Therefore we provide the **NodeInfo** class that handles the node information. Notice this class is not the same as the **Node** from the **CNc** diagram. The **CNa** is associated to the node through the **Driver** class.

The **CNa** is responsible to instantiate one **Driver** object per managed device to help handle the interaction with nodes. Which implementation to choose is defined by a configuration

file provided on startup for each one of the managed nodes. The `devDriver` interface defines the common interface for the plugin and the `DriverConf` provides the necessary operation configurations.

Finally we can see that `CNa` implements the `devConf` interface, since this is the interface through which the `CNc` will provide information to the agents.

4.3.2 Databases

Traditionally it would be normal to somehow separate the design of the data model from the object model, and then specify a data access layer to provide the access to the persistence mechanisms. With the use of a persistence framework (like the Python SQLAlchemy), we can directly use our object model and be abstracted from the DB details. Therefore our data model will basically be a subset of the object model presented above.

As discussed before, we will keep our database as simple as possible, avoiding duplicated information already available on the cloud framework. As we'll see on the data model, we only store some information about the virtual networks, because we need to associate links and ports to them. Another important detail is that only the `CNc` will access the DB directly. All the information necessary to the `CNas` will be provided through its API, i.e. the `devConf` interface.

On figure A.3 (appendix A) we present the proposed model for the `CNc` database.

4.3.3 External APIs

The APIs provided by both, the `CNc` and the `CNa` components are basically defined as a subset of the interfaces already presented before.

For the final users, i.e. the managers, the `CNc` component have three interfaces composing its APIs: the `vNetConf`, the `pNetConf` and the `CNcMgr`.

Any manager that registers to obtain updates must implement at least one of the `vNetMon` and `pNetMon` interfaces, depending on the updates it wants to receive.

There is also another interface part of the `CNc` API: the `devMon` interface, that enables any `CNa` to report updates of the device state.

Finally, the `CNa` API is basically composed by the interface `devConf`.

4.3.4 Command Line Interfaces

The manager responsible to handle the virtual network deployment will be the cloud framework. Therefore will be its responsibility to implement the CLI for virtual network related operations. But the cloud framework won't handle the physical network component. Therefore we need to propose a CLI for the physical network manager. This is a very important aspect of the solution design since it gives the end user an interface to the system. The CLI for the OpenStack component is also described on the next section.

Following are a list of commands that should be implemented by the CLI and the respective description.

| | |
|-------------------------------------|---------------------------------|
| <code>neutron node-create</code> | Create a Node |
| <code>neutron node-delete</code> | Delete a given Node |
| <code>neutron node-update</code> | Update a given Node |
| <code>neutron node-list</code> | List all the Nodes |
| <code>neutron node-show</code> | Show details of a given Node |
| <code>neutron node-attach</code> | Attach a Node to a Segment |
| <code>neutron node-detach</code> | Detach a Node from a Segment |
| | |
| <code>neutron segment-create</code> | Create a Segment |
| <code>neutron segment-delete</code> | Delete a given Segment |
| <code>neutron segment-update</code> | Update a given Segment |
| <code>neutron segment-list</code> | List all Segments |
| <code>neutron segment-show</code> | Show details of a Segment |
| | |
| <code>neutron port-update</code> | Update information about a Port |
| <code>neutron port-list</code> | List all the Ports |
| <code>neutron port-show</code> | Show details of a Port |
| | |
| <code>neutron link-update</code> | Update information about a Link |
| <code>neutron link-list</code> | List all the Links |
| <code>neutron link-show</code> | Show details of a Link |

4.4 FRAMEWORK OPERATION

Before detailing the integration with OpenStack it might be useful to understand how the system works and mainly how both systems are orchestrated to provide the service to the end user. We won't provide a detailed description of all the possible operations because they would be very similar, so we will give a summary and in the meantime we provide detailed descriptions of a couple of important operations, that will give the reader a very good idea of how the whole system works.

4.4.1 Physical network management

For the physical network management OpenStack is not involved. We have two types of events: user initiated events and events reported by the agents.

User initiated actions can all be summarized in one operation: update network topology. When an update to network topology occurs there's two base actions triggered in the CNc: update the DB and remap the virtual networks. After that any manager that is registered is notified about possible changes that occurred and the operation is over.

The remapping of the virtual networks may be an expensive process to execute every time a physical network change occurs. Therefore the execution of this operation is optional and controlled by the user who performs the action. There's three possibilities: (1) the remapping

is executed for all networks; (2) the remapping is executed only for virtual networks for what after the operation at least one of the ports doesn't have connectivity; (3) the remapping isn't executed at all, even if there are ports that will loose connectivity.

The second class of events are the ones reported by the multiple CNas. The actions triggered on the CNc are nonetheless the same. The DB is updated, the network is (eventually) remapped and the managers are notified. The remapping of the virtual networks here is not controlled by the operator, since there is no operator, but by a configuration provided to the CNc.

4.4.2 Virtual network management

In terms of data management, the virtual network operations are very similar to the physical ones. The operation is executed, changes are stored in the DB, and managers are eventually notified. With a small difference though, the virtual network mapping is triggered only for the affected network, if any. Some operations, namely operations to create nodes or segments, or updates that affect only devices not currently involved in any virtual network, don't trigger any network remapping.

One thing we didn't discuss until now is how the virtual networks are mapped. The mapping consists of two parts: (1) the mapping algorithm is executed resulting in the creation of three sets of links: new links, links to update and links to delete, and (2) the links are provided to the respective agents to be deployed on the network.

In case of a failure deploying a link it can be configured whether the CNc should keep trying to remap the virtual network or should just give up.

The deployment of a virtual network (or changes to a virtual network) is not an atomic operation, since that would mean that all the virtual network would be down in case of failure deploying a single link. However there are some special cases to consider, namely when remapping a network. When the failure occurs on the deletion or update of a link, all the changes should be reverted, since these failures could potentially lead to loops in the virtual network. If the failure is in the creation of a link, the deployment can proceed and eventually the network can be remapped to find a solution to the missing links.

4.4.3 Interoperation with OpenStack

The integration with OpenStack is probably the point that may need the more thorough explanation. Therefore before describing the necessary changes to the cloud framework we will give a detailed description of how the components should interact among them by giving a detailed description of two of the most complex operations. Other operations follows basically similar steps.

The first operation is the creation of an external port. This is an user initiated operation, therefore OpenStack should receive the request through its API. OpenStack will then use the CNc API to request the creation of a new port on the specified device. This last operation is executed by the ML2 mechanism driver that should know how to communicate with the CNc. The control passes then to the CNc component that will check the operation arguments,

namely if the device exists. In positive case the correspondent CNa is informed about the request and will try to effectively deploy a new port on the device. If the operation succeeds, then a new device specific identifier will be return by the CNa to the CNc and then to the OpenStack. They both register the port id on their databases. If the operation fails in any point, then no action takes place and the user is informed. The most common reason for this operation to fail will probably be the device running out of ports.

Now the most complex operation is probably the attachment of an external port to a virtual network. The process starts again with the user requesting the attachment. After the request, the ML2 mechanism driver is informed of the change and will verify if the virtual network already has any external port attached. In positive case, the driver will proceed with the request to the CNc. The CNc will then check the request and if valid proceed with a virtual network remapping. This is now a virtual network operation similar to but similar in terms of steps to the creation of a port we described above. In case this is the first port, before do the request the ML2 driver will need to request the creation of an external port in every compute node, and the attachment of the newly created port to the virtual network. Only in case these operations succeed, the driver will effectively request the attachment of the external port requested by the user.

4.5 OPENSTACK INTEGRATION

As discussed before the guidelines here are to change OpenStack as little as possible. And the CNc interfaces were designed with that in mind. OpenStack will basically only know about the ports that are currently part of their virtual networks. And it is not even necessary that OpenStack keeps any state about those ports since it can always be retrieved from the CNc when necessary (OpenStack can cache that state though). Ports are requested from nodes. So, in order to provide the user interface, OpenStack may also want to know about nodes. But once again, it doesn't need to keep any information about the nodes existent on the physical network since that information must be updated from the CNc every time it is necessary to OpenStack.

OpenStack already has the concept of port. This port represents a virtual entity through which virtual devices access the network. These can be VMs, but also virtual routers, DHCP servers, etc. With the introduction of our framework, the concept of port is extended representing its external counterparts. One of the possibilities to integrate the new concept on the OpenStack framework would be to extend the OpenStack ports to also represent our physical ones, but there are some incompatibilities. To avoid confusion lets call port to the OpenStack ports, and external port to our campus network ports. Unlike what happens with external ports, ports are associated with a specific device (whatever the device is), this means they have associated Media Access Control (MAC) and IP addresses, the associated device/device-id represents not the node that provides the port but rather the device to which the port is attached, there are also associated security groups, and all of the management must

be different considering the different nature of each one. Therefore the concept of external port is introduced in the OpenStack framework as a new entity.

All of the extensions proposed to OpenStack will in fact be around the management of these new external ports. As we already explained, a list of ports (external ports in this context) can completely describe the virtual network topology, therefore it is the only necessary component. About the nodes, OpenStack will basically only need to handle their name and associate each port with the name of the device that provides it, and in fact, it's only an user interface requirement.

The integration with OpenStack will comprehend three major aspects:

1. Update OpenStack API, DB model and CLI tools to support the new concept of external port;
2. Implement the communication with CNc including the callbacks for updating external ports state;
3. Provide a mechanism that provides the connection between the virtual and physical environments.

One of the major aspects we need to consider when referring to the OpenStack integration is that traditional virtual networks must keep working alongside the new campus networks, i.e. standard virtual networks can still be deployed without taking care of the new external ports. During the Havana development cycle a new core plugin was developed with the purpose of permitting multiple types of networks (previously implemented by different plugins and therefore mutually exclusive) to be deployed simultaneously. Considering that (1) it's predictable this becomes the core plugin after the Havana release, and (2) that the plugin was design to be extended, this is the obvious choice to implement our extensions.

About the first integration point the API and CLI changes are pretty straight forward. The user should be able to retrieve the available ports with their state, request the creation of new ones, and associate and dissociate ports with virtual networks. On section 4.5.2 these changes are properly detailed. The data model changes are also very easy to implement since OpenStack doesn't have to keep a lot of information about the physical network. The only required change is the addition of a new entity called **externalPorts** as shown on figure 4.1 that described the proposed extension to the OpenStack base data model.

The second point is also very straight forward. OpenStack must implement the CNc APIs in order to communicate with it. Those APIs will be used when the user performs an external port related operation or, in the case of the callbacks, when there is a change on the virtual network. There is not a lot more information we can give here since both the interfaces and the communication among OpenStack and CNc were already described on previous sections.

Finally the last point requires the major discussion. There is multiple ways to provide the connectivity among the virtual and physical environments. But one of the main objectives is that OpenStack and our framework be completely independent, as so, neither OpenStack should manage the deployment of the external ports, nor our framework should know how

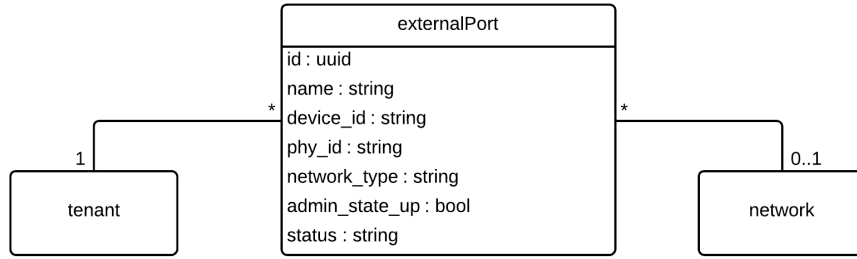


Figure 4.1: Proposed extensions to the OpenStack data model

OpenStack works internally. The solution found is the creation of an OpenStack aware device driver. Compute nodes will be considered nodes and is the responsibility of OpenStack to request the creation of external ports on the necessary compute nodes, i.e. when a virtual network has external ports attached OpenStack will take care to request CNc that an external port is created on every compute node were that virtual network is relevant. The necessary information for the driver to attach the external port to the virtual network is provided on the additional port properties we considered before. This way only the driver knows how both frameworks work what is not a problem since drivers are pluggable. The CNc and mapping algorithms don't need to be changed to accommodate OpenStack, since from its point of view the port created is just a standard port like any other (and in fact it is, since it meets the requirements).

One last thing that should be noticed about the integration plan is that the introduction of an external port concept doesn't bind OpenStack to our framework. Our framework should be seen as one of the options to implement external ports, but not the only one. It is possible to have external ports that are simply tunnels to a remote location, or the creation of a VLAN. So we will try to keep OpenStack, and the external port as generic as possible to enable different external port implementations in the future.

4.5.1 ML2 integration

The ML2 plugin was designed to be extensible. This plugin introduce two new entities responsible for the management of the network. The first one is the Type Driver: this is a class responsible to manage type related information, e.g. assign VLAN IDs when using VLANs, register the tunnel IDs when using GRE, etc. The second entity is the Mechanism Driver. This entity is responsible to handle the interaction with external network controllers and inform them about changes to the network, like the creation of a port or the update of a network, for instance. Therefore, the Mechanism Driver is the ideal point of extension to implement our changes.

Currently, the mechanism driver interface has methods to handle Create, Read, Update and Delete (CRUD) operations on networks, subnets and ports. It is therefore proposed to extend its interface with methods to handle external port operations. Follows a list of the

new methods. Notice that for each operation two methods are available, one is called by the ML2 plugin before the changes being persisted in the database, and should just check that the operation can proceed. The second one is called after the commit of the information and should effectively request the necessary changes to the network devices.

```
create_external_port_precommit(self, context)
create_external_port_postcommit(self, context)
update_external_port_precommit(self, context)
update_external_port_postcommit(self, context)
delete_external_port_precommit(self, context)
delete_external_port_postcommit(self, context)
```

A new class `ExternalPortContext` will also need to be created to carry the necessary context for the external port operations.

To effectively implement the operations each network type that supports external ports should implement a mechanism driver that implements the above operations. The mechanism driver is the responsible to request the creation of the external ports on the compute nodes, that were referred on the previous section.

Beyond the mechanism driver changes, the ML2 plugin will also have to be extended to handle the new API we will detail next.

4.5.2 APIs extensions

To support the new external port concept, OpenStack API needs to be extended with the new entity and a set of operations on top of it. On table B.1 (appendix B) we can find a detailed specification of the new `ExternalPort` entity.

Next we describe the new operations available for the external port entity. A summary is provided on table B.2 on appendix B.

Create external port. This operation requests the creation of an external port on a given device and associates it with a given tenant. If the device is able to provide the new port it returns an identifier (`phy_id`) that is then associated with the port. Otherwise the operation fails and the port is not created.

Update external port. The update operation is used to:

- Update the external port name
- Turn the external port up or down (update the `admin_status`)
- Associate the external port to a given tenant
- Attach an external port to a given virtual network
- Detach an external port from a given virtual network

The first and second operations have no special constraints. It's only important to notice that the device can refuse to update the admin status of a port, in which case an error is returned.

The third and fourth operations must ensure that the port is not associated with a network belonging to a tenant different than the one that owns the port. The last operation has no special constraints.

Delete external port. Request an external port to be deleted from a device. A port cannot be deleted if it is currently attached to a network.

List external ports. List the external ports owned by the tenant that submits the request. According to the policy if the user has admin rights all the available ports are listed.

Show external port. Returns the information of a specified external port. Only admins or the user that owns the port can see its information.

4.5.3 CLIs extensions

The command line interface for neutron is developed as a separate project. It directly uses the API to execute the operations. Following is a list of the new commands that should be added to the tool.

| | |
|---|---|
| <code>neutron extport-create</code> | Request the creation of an external port on a given device. |
| <code>neutron extport-update</code> | Update the attributes of an external port. |
| <code>neutron extport-delete</code> | Request the deletion of a given external port. |
| <code>neutron extport-list</code> | List the external ports belonging to a tenant. |
| <code>neutron extport-show</code> | Show informations about a given external port. |
| <code>neutron extport-attach</code> | Attach an external port to a given virtual network. |
| <code>neutron extport-detach</code> | Detach an external port from a given virtual network. |

4.6 EXPERIMENTS AND THE EVOLUTIONARY VIEW

Although we do not provide an implementation as part of this work, the development of the conceptual solution was accompanied with implementation tests. Deploying the framework and hack some of its components provided us with a very good insight that then allowed to write this document. This section gives the reader a description of the executed tests and some of the problems found.

On the begin of the project the idea was to find a quick approach (or more like a quick hack) that would enable the connection of virtual networks to physical ones. Multiple ideas

came out, but the chosen one was the implementation of a provider router. The provider router concept, as we thought about it, represents a physical or virtual router, external to OpenStack, that would be used by the user as if it was a traditional virtual router, but that would externally be connected to the physical networks of interest. This approach would let the users keep using its traditional interface, and require few changes to the system.

So we started looking into OpenStack's Neutron component (Quantum at the time), and mainly at the OpenVSwitch (OVS) plugin to check how could it be changed to allow the connection to external routers. Although the framework is very modular, the same doesn't really apply to the plugins themselves. OVS plugin expects that its standard setup is there and doesn't allow for much configuration. We did a couple of tests with the purpose of setup a GRE tunnel to a linux machine when a provider router was attached to a network. We also implement some modifications to the Quantum CLI to enable the user to perform these operations.

Based on the experiments before, the BP Provider Router Extension was published. The proposed solution is detailed on [2], but the BP didn't had a great acceptance from the community, and as it doesn't really provides a complete solution we started investigating other options.

We started then developing a solution (the one we presented) that completely fulfill the requirements. During the development we did some small tests to check whether some of the changes were possible or how hard will it be to implement them. Mainly related to the integration with OpenStack.

Luckily, during the Havana release cycle, the ML2 plugin started to be developed. This plugin promised to resolve the problems we had with the OVS one, given it was developed to be extensible. We then start to think how to integrate the solution we envisioned with OpenStack, we looked mainly on how to extend this plugin, the OpenStack CLI and API, and came up with a partial implementation of the "ML2 External Port" BP, detailed on [4].

Chapter Five

Conclusions

The need to improve the university's IT services is obvious. The current model not only represents a waste of resources but is also conditioning the deployment of new approaches to teaching, and the development of new projects. The framework proposed on this work represents a step forward for the provisioning of the networking services in the university campus.

The proposed solution makes use of the advantages brought by the new cloud frameworks and provides an integrated management of the physical and virtual worlds. Although targeted at OpenStack, the solution and concepts presented are independent of the cloud framework and can be easily ported to another solution or even used standalone.

Although we properly described the solution with all the necessary details, we still miss an implementation, to effectively measure its success.

5.1 FUTURE WORK

The most obvious point is to provide a working implementation of the framework. Only then we will be able to effectively analyse the solution and understand its strengths and weaknesses.

Even though we don't have an implementation yet, there's already some points we can define as important future work.

One of the most crucial points not discussed in this work is the segregation between the definition of the virtual network and its effective deployment. In its current state, changes are deployed as soon as they're specified, meaning that jumping among two different layouts means to define the layout every time. It would be useful to define a network topology, save it on the user workspace and then load it to the network, replacing the current state.

The possibility of defining constraints for the virtual network deployment is another possible working item. Its possible that some virtual networks have specific requirements like security, bandwidth or others. Provide means for the specification of these requirements would give the user even more power to specify how their networks are deployed.

Getting to the vision of a fully virtualized campus is also in the list for the future work. Now we have a solution for the networking component it's time to think about the integration

of other services. A possible next step is to integrate with the OpenStack Ironic project. The Ironic framework brings the flexibility of cloud to bare metal computing, letting the user work with physical computers the same way as with virtual ones. Integrate this service in the overall solution would enable the seamless migration of not only networks but also machines across the campus.

Appendices

Appendix A

Architectural diagrams

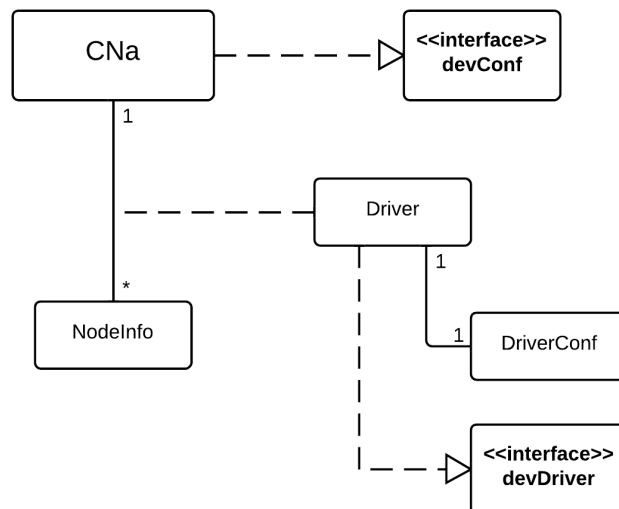


Figure A.2: Class diagram for the CNa component.

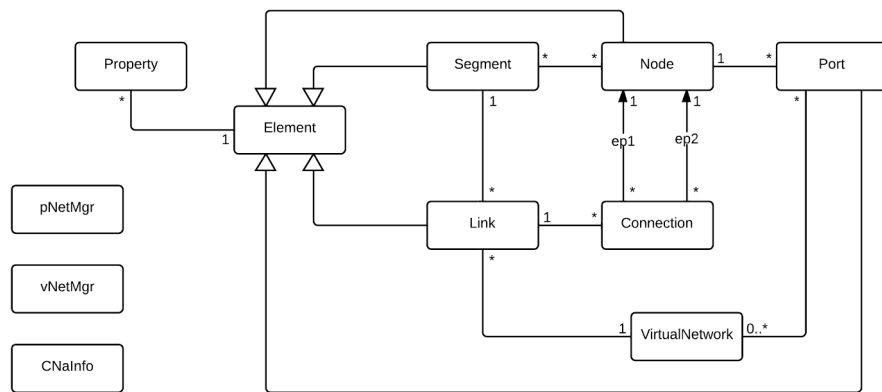


Figure A.3: Campus Network Controller Database scheme

Appendix B

OpenStack extensions

Table B.1: ExternalPort API attributes

| Attribute | Type | Required | CRUD | Default value | Validation constraints | Description |
|----------------|----------|----------|------|---------------|--|---|
| id | uuid-str | N/A | R | generated | N/A | UUID representing the external port. |
| name | String | no | CRU | None | No constraints | Human-readable name of the port. |
| device_id | String | no | CRU | None | Non-empty string | Device providing the external port. |
| phy_id | String | N/A | R | None | N/A | Port identifier attributed by the device. |
| network_type | String | yes | CR | N/A | Any available network type | Type of network responsible for the instantiation of this external port. |
| admin_state_up | Bool | no | CRU | True | True False | Administrative state if the external port. |
| status | Status | N/A | R | N/A | N/A | Status of the external port: <ul style="list-style-type: none"> • ACTIVE • DOWN • BUILD • ERROR |
| tenant_id | uuid-str | yes | CRU | N/A | No constraints | UUID identifying the tenant holding this external port. |
| network_id | uuid-str | no | CRUD | None | Existent network ID belonging to the same tenant as this port. | UUID identifying the network this external port belongs to if it exists. |

Table B.2: API operations for external ports

| Operation | Verb | URI | Description |
|-----------|------|-------------------|---|
| Create | POST | /externalports | Request te creation of an external port on a device. |
| Update | PUT | /externalports/id | Updates an external port. |
| Delete | DEL | /externalports/id | Request the deletion of external ports belonging to the tenant. |
| List | GET | /externalports | Lists the available external prots belonging to the tenant. |
| Show | GET | /externalports/id | Shows informations about the external port. |

Appendix C

Blueprint

Provider Router Extension

Quantum Provider Router Extension

Scope

Specify a Quantum API extension that provides a way to map non-OpenStack routers in the Quantum database and provide them to tenants as virtual ones with complete API support.

This extension will provide to routers similar capabilities to what the provider network extension provides to networks, i.e. directly map and use physical resources on demand as virtual ones.

Integration of new private clouds on legacy infrastructures and on demand connection between virtual and physical resources will be then much simplified.

Use cases

Currently, the integration of physical components in the virtual environment is of vital importance to allow the migration of currently deployed infrastructures to new private clouds. The main use cases will be related with connecting physical to virtual private environments through a legacy network that cannot be easily changed.

Following is a list of example use cases:

- Connect a virtualized environment to distant physical networks on demand
- Have physical hosts on a private physical network being able to reach VMs on a private virtualized environment

Implementation Overview

There are three main points to consider for the implementation of this extension:

- Who is responsible to communicate changes to the physical routers
- How to communicate with physical routers
 - This should be extensible in order to support any router
- How router interfaces will be connected to tenant networks
 - The traffic between the VMs and the physical routers should be isolated

For the first point, the communication with the physical routers will be of the responsibility of the L3 agent. When a operation is executed (e.g. `router-interface-add`) the agent will check if the router is a provider router and, in case it is, communicate that change to the correspondent physical router.

For the second point, the management of the physical routers will be done through a driver implementing a generic interface. This will keep the software easily extensible and enable the use of any equipment from a linux box to a carrier grade router by implementing the respective driver. The driver implementation will be up to it's creator and dependent on the physical router characteristics. It can use a management interface, through telnet or ssh, the snmp protocol, or any other mechanism that is appropriated.

For the third point, the technology used to connect the router to the tenant network (e.g. GRE) must be specified in the API and must be supported both by the L2 plugin used in the compute nodes (e.g. OpenVSwitch) and the physical router.

Physical networks

As described above, provider routers can be connected to physical networks, only accessible through them. To support this a new provider network type is proposed: `PHY`.

This is a type of network not directly accessible to the compute nodes, but accessible through some device that support this network type. This device can be a provider router or any other device that can be implemented in the future.

The implementation of this type of network will require changes to the quantum plugins. Support will be implemented, at least, on the openvswitch one.

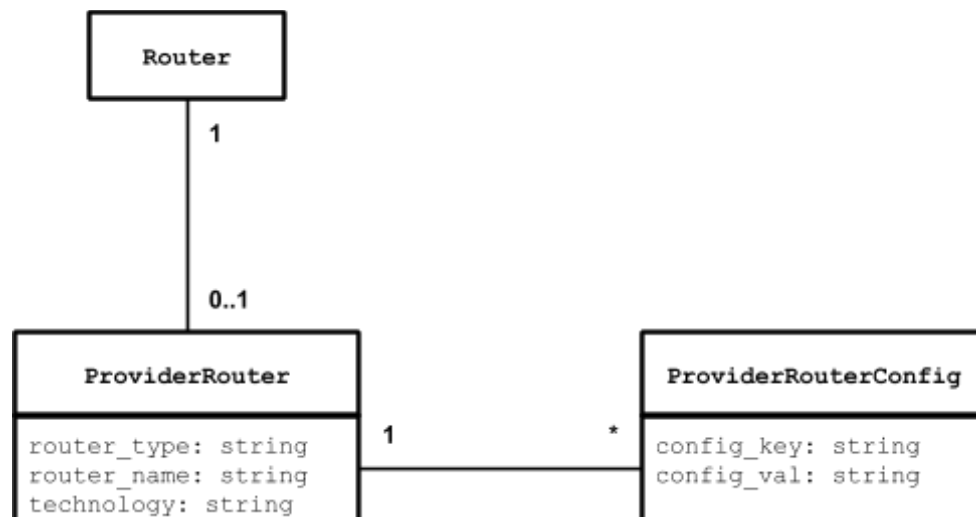
Implementation plan

The implementation of this extension will roughly follow the next plan:

1. API modifications
2. Changes to the database model and related code
3. Sketch the driver interface
4. Implement support on the L3 agent
 - a. Tenant network support
 - b. `router:external` network support
 - c. Floating IPs support
5. Perform the necessary changes to the plugins to support `PHY` networks

Data Model Changes

The database model extension is shown in the diagram below. It adds two new tables to the quantum database that will hold the provider configuration, for the provider routers.



Configuration Variables

A new configuration variable is necessary to provide the mapping between the `router_type` and the correspondent driver. The variable `provider_router_driver_mapping` will go to the `l3_agent.ini` file.

API's

The provider router extends the Quantum API by creating new attributes. No new resources or actions are defined.

Extended attributes

The router structure will be extended with 4 new attributes in the provider namespace:

- **provider:router_type** [CR]:
Specifies the type of router. This will be used to choose the driver that will handle the communication with the physical router.
- **provider:router_name** [CR]:
Specifies the router name. This is the way the driver will identify which physical router it is configuring.
- **provider:technology** [CR]:
Specifies the technology used to place physical router interfaces on tenant networks (e.g. GRE).
- **provider:router_conf** [CRU]:
Any number of key/value pairs used by the driver to configure the router. Those can be the router IP address, a credentials file, or any other, dependent on the driver (i.e. the **router_type**).

Authorization

By default provider routers can only be created or deleted by admin users. The provider attributes are also only visible, and can only be updated by admins.

This routers can nonetheless be assigned to any tenant. The tenant will then be able operate the router the same way it operates a virtual router. The only exception is **PHX** provider networks, which can only be attached to the router by an admin user.

Plugin Interface

This blueprint introduces changes to the `quantum.agent.linux.LinuxInterfaceDriver` interface. The methods will receive a new parameter **provider**, a dictionary containing the provider configuration for the router. To keep compatibility, the parameter will default **None**.

Required Plugin Support

As explained above a small change in the quantum plugins is required to support the new provider network type. At least the openvswitch plugin will be modified to support this.

L3 related code (l3 agent, l3 db, etc) will be modified to support this extensions.

Dependencies

Quantum components

This extension depends on the quantum L3 extension (router).

Python dependencies

The extension itself doesn't have any specific requirements. However it is possible that the used drivers can introduce new requirements. It is expected that to manage the routers, drivers will use telnet, ssh, snmp and other libraries.

New Quantum blueprints

There are some blueprints currently in development, targeting the Havana release, that will impact the implementation of this blueprint.

Migrate L3 router service from mixin to plugin

<https://blueprints.launchpad.net/quantum/+spec/quantum-scheduler>

This blueprint moves L3 functionality to a separate plugin. Implementation will take that in consideration, but it doesn't entails many differences since the changes will still be done mainly to the `L3NATAgent` and `L3_NAT_db_mixin` classes.

Multiple L3 and DHCP agents for Quantum (Implemented in Grizzly)

<https://blueprints.launchpad.net/quantum/+spec/quantum-scheduler>

The most direct consequence of this blueprint is that the physical router will be connected to multiple compute nodes instead of just one network node.

Quantum Multi-host DHCP and L3

<https://blueprints.launchpad.net/quantum/+spec/quantum-multihost>

Same as above.

Make Authorization Orthogonal

<https://blueprints.launchpad.net/quantum/+spec/make-authz-orthogonal>

Implementation will conform to the improved authorization mechanism.

CLI Requirements

The `router-create` and `router-update` operations will need to support the new provider attributes. The quantum client application already handles this, so no changes needed.

Horizon Requirements

The create router dialog box will need a new tab to permit the introduction of the provider attributes. These attributes are not mandatory and should only be provided on the creation of provider routers. Configuration values should be provided on a text box, one per line.

Usage Example

Creation of a provider router with the quantum client will look like:

```
quantum router-create r1 \  
    --provider:router_type abc \  
    --provider:router_name r1.local \  
    --provider:technology gre \  
    --provider:config list=true creds=/etc/quantum/hw-creds/r1
```


Beware that `list=true` is always necessary to `--provider:router_config` when you provide only **one** key/value pair, otherwise it is not necessary but can be used.

Test Cases

New unit tests will be added to test the manipulation of provider routers. The following components must be tested:

- Database manipulation code for provider attributes
- L3 agent operation with provider routers
 - Internal networks
 - External networks
 - Floating IPs
- Manipulation of the new network type `vxr` by plugins

As a physical router is not available for testing, a dummy driver can be used to check the correct operation of the L3 agent.

Appendix D

Blueprint

Campus Network Extension

Neutron Campus Network Extension

Scope

Extend Neutron with mechanisms to support the virtualization of a physical heterogeneous network infrastructure. This will extend virtual networking boundaries outside the datacenter supporting advanced scenarios on top of any network deployment.

This extension proposes mechanisms to automatically create overlay L2 networks by the direct control of heterogeneous networking equipment using multiple underlying technologies. This overlay networks will connect virtual hosts on a datacenter running OpenStack to physical hosts on geographically separated network segments outside the datacenter. Creation of the overlays will take into consideration the current network deployment allowing the integration of this new services on top a legacy network without service disruption.

The Campus Network extension is **mostly** targeted at private cloud scenarios as it allows the integration of the so called fog in the cloud computing scenario which aren't usually available on public clouds, but are of much value to enable the development of advanced scenarios on private cloud deployments.

Use Cases

Consider the network sketched on figure 1 below. The following use cases are examples of scenarios automatically deployed on top of that physical network using the Campus Network extension. All the network equipment configuration necessary to implement these scenarios are automatically executed by Neutron in accordance with specified configurations.

1. Multi-segment network

The main purpose of this extension is to join on the same L2 domain a set of network segments, dispersed over the physical network, and a set of virtual machines running on OpenStack, possibly on different compute nodes.

Consider figure 2 to be the target deployment. The servers are virtualized, running on the datacenter and the physical hosts are connected on two switches somewhere on the network. Mapping the components to their physical location on the network infrastructure we have the orange hosts on figure 1.

To achieve this use case, Neutron will automatically configure all the necessary equipments across the network, comprehending the configuration of switches *s1* and *s2*, the compute nodes *c1* and *c2*, and the router *R2*. To achieve traffic isolation (i.e. the creation of a virtual link across the network) multiple technologies can be used, e.g. VLANs on the switched/trunk network, and GRE tunnels inside the datacenter. The technologies in use are completely configurable and can take in consideration the constraints imposed by the current deployment (e.g limit the allocable VLAN IDs to the ones currently not in use on the network).

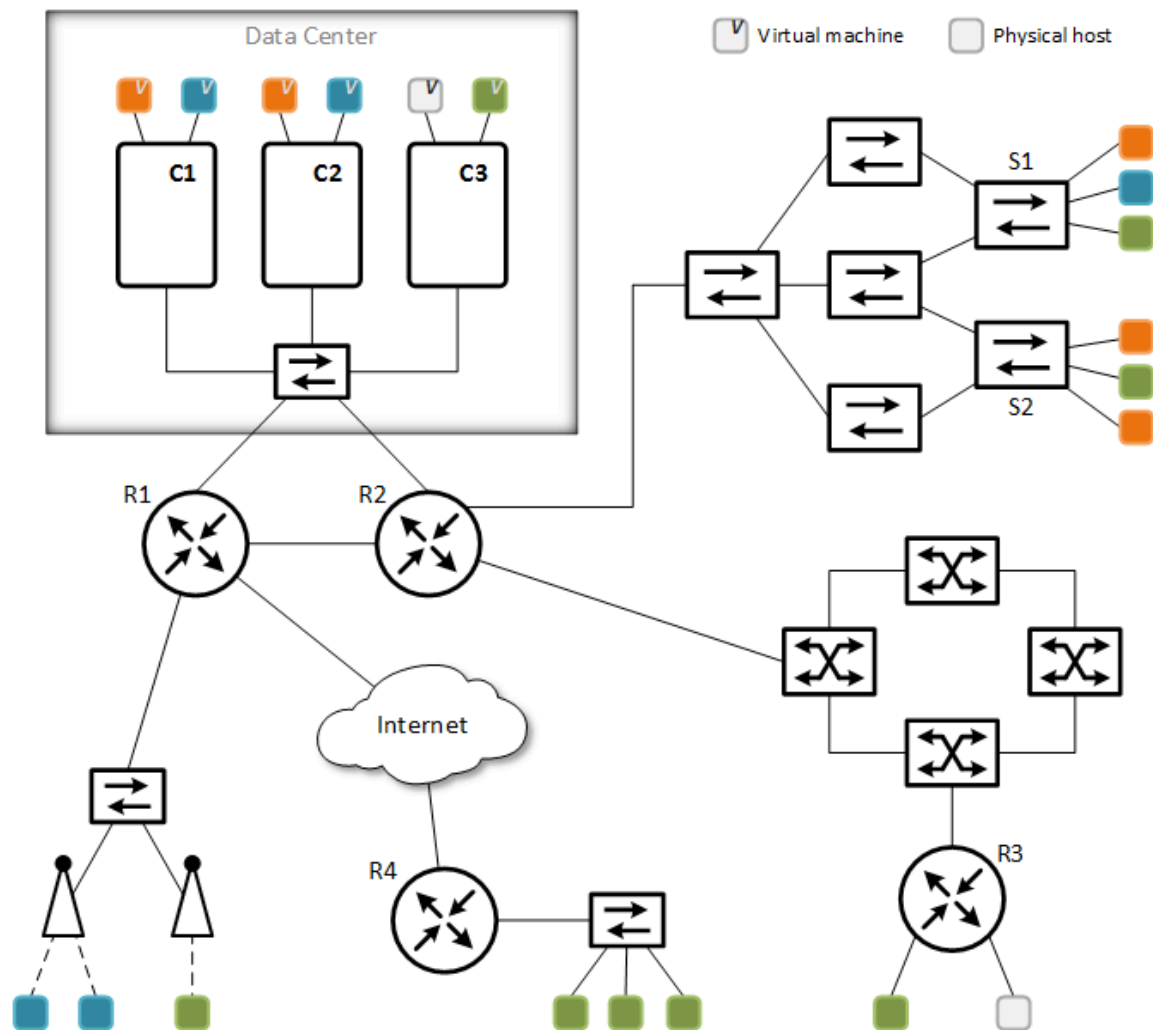


Figure 1 Physical network deployment

Later on, components can be added or removed from the virtual network topology, and all the necessary configurations will be done taking in consideration the current scenario (e.g. consider a VLAN ID is already allocated to this virtual network on the trunk segment).

The specification of the virtual network is all done through the extended Neutron API.

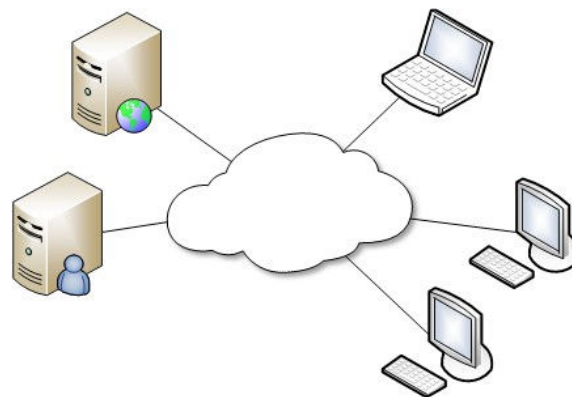


Figure 2 Virtual network required by the use case 1

2. Creation of access points

Consider now you have a demonstration scenario consisting of a couple of virtualized servers, and you want to provision a way to place guests' laptops on the same L2 domain as the servers to demonstrate your new service. Figure 2 can represent this as well.

To enable this use case, Neutron can automatically configure a new SSID on one or more access points so that all the hosts connecting to this new SSID will automatically be on the same virtual L2 domain as the servers. Specific ports on managed switches can also be configured to belong to the same virtual network as well as with any other technology you might want to use.

To deploy this network some other equipments also need to be configured similarly to the ones of the first use case, like the router Σ . Of course these configurations will also be automatic.

This scenario is represented by the blue hosts on figure 1.

3. Cross the internet

As you can notice in the previous use cases, multiple technologies can be crossed by the same virtual L2 network. The nodes in green on figure 1 represent a more advanced scenario crossing even more technologies.

Being able to configure any equipment using any technology makes crossing the internet or any other domain that you don't control a possible use case, using for instance tunnels or a VPN technology of your choice. Neutron will take care of all the necessary configurations and everything is automatically provisioned. From the user point of view, it is only necessary to specify which segments will connect to the virtual network, using the extended Neutron API.

Implementation Overview

The final purpose of this blueprint is to automatically deploy virtualized networks on top of an heterogeneous deployment in a non-disruptive manner. Its implementation brings a lot of questions both conceptual (e.g. how to handle the physical network complexity, how to map the virtual network to the physical substrate) and implementation related (e.g. define the API extensions, how to keep the current implementation working in parallel, how to manage physical devices). These questions are addressed below starting by the conceptual definitions and then implementation related considerations.

Physical network abstraction: conceptual model

This extension enables on demand configuration of physical networks of any complexity. In order to reduce the complexity of the software and make the system more generic an abstraction of the physical network was created.

Four new concepts are introduced: `cSegment`, `cPort`, `cNode` and `cLink` (the prepended `c` is intended to distinguish these concepts from the generic concepts with the same name). These four entities should be enough to describe any network that can be deployed using this extension. Every operation will then be defined in terms of basic operations using these entities, e.g. `cSegment-create`, `cPort-request` or `cPort-associate`.

`cSegment`

A cSegment represents an opaque network segment able to route isolated L2 traffic, i.e. that supports the creation of virtual L2 links using some technology. A set of switches interconnected by trunk ports, a small L3 network and the internet are all valid examples.

An Important characteristic is that cSegments' complexity is hidden. From the point of view of Neutron a cSegment is characterized solely by the technology it supports to provide isolation and the set of cNodes it is connected to. It is also important to understand that a cSegment doesn't mandatorily map to a physical segment, and a physical segment can be part of multiple cSegments. You can see a cSegment as a domain connecting a set of nodes (cNodes) able to use a specific technology to create virtual links among them.

The creation of virtual links can make use of any existing technology. This can be a more traditional technology, like VLANs or GRE tunnels, a more recent technology like VXLAN. Using openVPN or something like SSH tunnels is also possible.

cPort

A cPort is a point of access to the network. This entity can represent a lot of different physical realizations, it can be a port on a managed switch, an SSID on a wifi access point, a port on a virtual switch running on a compute node, or any other equipment/technology/identifier that gives virtual or physical equipments access to the network.

A virtual network can now be seen as an L2 domain giving connectivity to a set of cPorts. cPorts are available on cNodes where its configuration takes place.

cNode

A cNode is the entity responsible to provide cPorts and/or interconnect cSegments. Once again this can represent any equipment: a router, a wifi access point, a linux box, a virtual switch, etc.

The equipment represented by the cNode must be able to "translate" between the technologies used in the multiple cSegments it connects to and the cPorts it provides, e.g. switch traffic from a VLAN to a GRE tunnel or switch VLAN tags from a cSegment to the other.

cLink

A cLink represents the mapping between a virtual network and it's implementation on a specific cSegment. It is composed by an identifier and a set of connections among cNodes belonging to the cSegment. The identifier can directly map to the identifier used on the network (e.g. VLAN ID) or can be just a logic identifier.

Each cSegment provides a set of cLinks available for allocation.

Operation overview

The operation of this type of networks requires two different sets of actions, i.e. two different interfaces. The first interface is intended for administrators to specify the physical network layout and the second interface is intended for the network controller to manage the network.

A cSegment represents an opaque network segment able to route isolated L2 traffic, i.e. that supports the creation of virtual L2 links using some technology. A set of switches interconnected by trunk ports, a small L3 network and the internet are all valid examples.

An Important characteristic is that cSegments' complexity is hidden. From the point of view of Neutron a cSegment is characterized solely by the technology it supports to provide isolation and the set of cNodes it is connected to. It is also important to understand that a cSegment doesn't mandatorily map to a physical segment, and a physical segment can be part of multiple cSegments. You can see a cSegment as a domain connecting a set of nodes (cNodes) able to use a specific technology to create virtual links among them.

The creation of virtual links can make use of any existing technology. This can be a more traditional technology, like VLANs or GRE tunnels, a more recent technology like VXLAN. Using openVPN or something like SSH tunnels is also possible.

cPort

A cPort is a point of access to the network. This entity can represent a lot of different physical realizations, it can be a port on a managed switch, an SSID on a wifi access point, a port on a virtual switch running on a compute node, or any other equipment/technology/identifier that gives virtual or physical equipments access to the network.

A virtual network can now be seen as an L2 domain giving connectivity to a set of cPorts. cPorts are available on cNodes where its configuration takes place.

cNode

A cNode is the entity responsible to provide cPorts and/or interconnect cSegments. Once again this can represent any equipment: a router, a wifi access point, a linux box, a virtual switch, etc.

The equipment represented by the cNode must be able to "translate" between the technologies used in the multiple cSegments it connects to and the cPorts it provides, e.g. switch traffic from a VLAN to a GRE tunnel or switch VLAN tags from a cSegment to the other.

cLinkerties. The second interface is intended for users to create and manage virtual networks on top of the physical substrate already specified.

Management of the physical network

The management of the physical network substrate consists basically of filling the Neutron database with information about the cSegments, cNodes, cPorts and cLinks available, their properties and the connections among them. Remember this extension isn't concerned about the physical network realization including the specificity of the network equipments, so this interface won't include methods to directly manage any device (like bring up interfaces, reset configurations, create a bridge, etc), only high-level operations to manage the four entities described above are considered. The details of the devices are to be managed by specific drivers and a per equipment configuration file.

Notice the specification of the physical network isn't static. Management operations are executed like any other operation with the services running, meaning physical entities can be created, deleted or modified somehow even if there are already virtual networks running. Every change to the physical network layout will trigger a remapping of the deployed virtual networks and eventually changes will be applied to the equipments.

Another important point to clarify is that Neutron is not responsible to check the physical network state or react to changes in the physical topology, e.g. if a cable is disconnected the network layout will not be updated deleting that connection. However, monitoring services could use the Neutron API to redefine the network layout and trigger a remapping of the virtual networks (although this is obviously out of scope here).

Deployment of virtual networks

From the point of view of the user, the deploy of a virtual network using this extension comprehends three main steps:

1. Create the virtual network (as already happens now);
2. Request cPorts (can be admin only);
3. Assign cPorts to virtual networks.

Notice that associating a cPort to a virtual network is splitted in two operations. First the cPort is requested and attributed to a specific tenant and then the tenant assigns the cPort to a specific virtual network. This enables a model similar to what currently happens with floating IPs, what eases the management of the physical resources letting admins specify which resources should be assigned to each tenant simply by making cPort requests an admin only operation.

User actions will trigger the system to map the virtual network specified to the physical network substrate and configure the physical devices accordingly, although this operations are transparent to the user.

From the point of view of the system, the first step consists basically in generating an UUID and register the network in the database, there are no configurations applied on the physical network. The second and third steps, on the other hand, lead to the actual work. When a cPort is assigned to a virtual network, Neutron will make sure every cPort on that virtual network can reach each other. This means realize what cSegments must be crossed and then apply the configurations to enable the traffic flow.

Something that is interesting to notice is that there are certain operations that implicitly requires a cPort to be created and assigned to the virtual network. Operations like booting a virtual machine or assigning a DHCP agent to the network are good examples. When one of these operations is executed the necessary sub-operations must be automatically executed.

Virtual network mapping

Every time a cPort is assigned to a virtual network the system will map that virtual network to the physical substrate in order all the cPorts can reach each other. The process consists basically in calculating a path across the physical network layout that connects all the cPorts that belong to the virtual network, specifying what cNodes are connected and how (through which cSegment). Depending on the physical network layout the solution may not be unique, what leads to the problem of defining what's the best solution, or at least, choose the solution to use.

The mapping algorithm can take a multiplicity of approaches to solve this problem. To improve the extensibility of the software mapping algorithms should be pluggable, similarly to what currently happens with schedulers, for instance. Some examples of what the algorithms may consider are whether to remap the entire network on every change or just amend the existing one, how to calculate the best path (e.g. static weights, segments' load), how the

network should be represented (e.g. using a graph) and what specific algorithm should be used to find the best path.

Regardless of the approach taken, the implementation of the mapping result is always performed in the same way:

- For each of the cSegments crossed by the virtual network
 - A cLink is requested for the virtual network
 - If a cLink is already assigned that one is returned
- For each pair of cNodes connected
 - Both cNodes are instructed to connect to each other via a cLink that crosses the predetermined cSegment
- For each cNode on the path:
 - The cNode is instructed to bridge a set of cLinks across the cSegments it connects to

A very important question to consider is the existence of loops on the virtual network mapping, because this is an L2 network and loops must be avoided. The use of protocols like STP is, at least for now, out of the question, for two reasons: (1) we cannot assume every equipment supports the used protocol and (2) the abstraction created makes this very difficult to specify. Since the mapping is centralized this is also undesirable because it is easier and more efficient to simply avoid loops. The used algorithm will have to take this into consideration.

There are however cases where it is impossible to avoid loops. This happens when the cLinks are not point-to-point being impossible to specify which cNodes are connected to each other inside the cLink. To exemplify consider figure 1 again: there are loops in the switches network that are impossible to be removed by Neutron since Neutron doesn't even know the physical network layout. This could be solved if beyond the configuration of the cNodes, cSegments were also configured by the system. To keep this blueprint simple this cases will not be considered, but the software design will take into consideration that later on, something like this can be implemented. For now the system simply considers that cLinks all provide point-to-point connections and is up to the administrator to manually configure the mechanisms to avoid loops on the necessary network segments. On a production deployment this is probably already done.

Finally the mapping algorithms may define new properties necessary to execute the mapping. This properties should be available in the database, as so, the database design (defined below) will have to take this in consideration and be easily extensible.

Mapping algorithm

As explained above, the mapping algorithm should be developed as a plugin. In order to keep this blueprint simple, here will only be defined a simple algorithm to execute the mapping.

The physical network will be represented on the system by a graph of connectivity between cNodes: a cNode is connected to every cNode it can directly reach through a single cSegment. Each edge is then associated with the cSegment that provides that connection and each vertice (cNode) is associated with the list of cPorts it provides. To exemplify, the network sketched on figure 1 is represented by the graph on figure 3 below (the cNodes are represented by the vertices; edges associated with the same cSegment are of the same color).

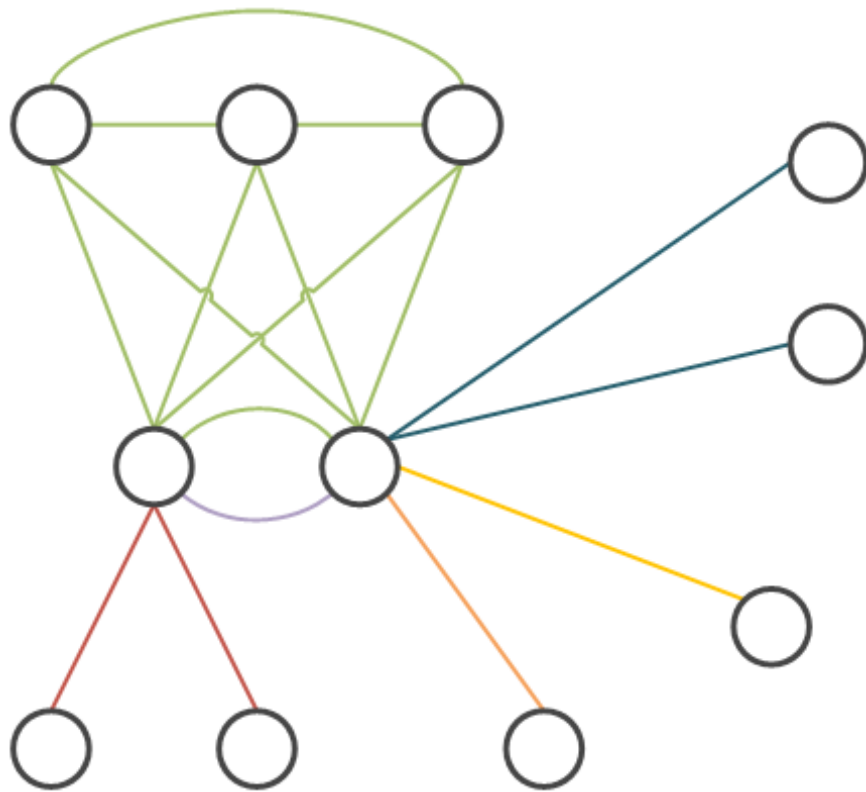


Figure 3 Graph representation of the network sketched on figure 1

Implementation architecture

Regarding implementation this extension will be divided in two main components: the **Network Controller** and the **Device Manager**.

The network controller will be responsible to manage the network as a whole and the device manager will be in charge of each physical device. The objective here is that the management of the devices can be segregated from the network controller. The Device Manager should be able to run anywhere, even in the device being managed, and can be responsible from one to multiple devices at the same time. To do so we need to strictly define each component responsibilities and define a communication interface.

Another key point to define is how the abstracted entities will be identified and mapped to its physical representation.

Network Controller

Network management will be centralized on the network controller. This component will be responsible to effectively implement the API and to execute basically all the operations related to this type of network, except configure the physical devices. This includes handle user commands, manage the database, map the virtual network to the physical substrate and trigger the configuration of the necessary resources. To execute the mapping of the virtual network this component will resort to a separate plugin as discussed before.

There should be only one instance of the network controller running per deployment.

Device Manager

The device manager will be the responsible to effectively apply configurations to devices. This is the component that knows how to communicate with the equipments and should

have all the informations to do so. Since each device is unique, a per-device configuration is necessary.

For the effective management of each device, the Device Manager will resort to drivers implementing a proper interface, what makes the Device Manager generic but able to handle any device. The driver should be specified per device giving the possibility to write generic drivers able to handle a specific equipment model, but also giving administrators the possibility of write drivers to handle specific devices with any specific constraint.

Device Drivers

The device drivers are the responsible to know how to effectively configure a device, from the communication mechanism to use (e.g. telnet, ssh, snmp) to the commands to execute. The device manager will instantiate a device driver per physical device it is in charge of.

Physical entities identification

To map between the abstract and physical entities, they will be named. The conceptual entities will have an user defined name that will then be used by the Device Manager to map the actions to the correct physical device/entity.

Integration with Neutron

A major concern on the implementation of this extension is giving tenants and administrators the possibility of keep traditional networks working the same way for the sake of simplicity and ease of use. Currently the ML2 plugin is the framework that better supports this requirement, as so it will be used as a base for the implementation of the Network Manager component proposed here.

The Network Manager will be implemented in terms of the ML2 TypeDriver and MechanismDriver. As this extension is intended to support a new type of network no changes should be done to the ML2 framework that affects the other network types. The implications to the ML2 framework are discussed below in the Plugin interface section.

About the data model, although there are some similarities the new entities proposed when the network abstraction was discussed have no matching entities on the current Neutron data model, as so, a data model extension is necessary and proposed below. Still, the existing similarities are worth discussing.

cPorts are similar to the existing ports in terms of specifying a point of access to the network. However ports are associated with a specific device, they are created on the creation of the device and have associated MAC and IP addresses, what doesn't match what happens with cPorts. cPorts can give access to one or multiple devices at the same time not being directly associated with a specific one, are usually limited in number (and mapped to a physical entity like a switch port) and it's creation is segregated from it's effective association with a virtual network. As so ports and cPorts will be considered different entities in the data model proposed.

cSegments can be compared to the segment concept introduced with ML2, but they are different concepts. A segment represents a more or less independent segment of a virtual network. cSegments are physical network pieces used to deploy virtual networks. A ML2 segment can consist of a virtual network deployed with this extension on top of multiple cSegments.

Data Model Changes

A set of extensions to the data model are necessary to support the requirements of this extension. The new entities proposed are defined on figure 4 below. There aren't any planned changes to the current database entities, only the addition of new ones.

The mapping algorithm in use will be able to store additional cSegment informations using the Properties class shown in the figure using a key-value pair scheme.

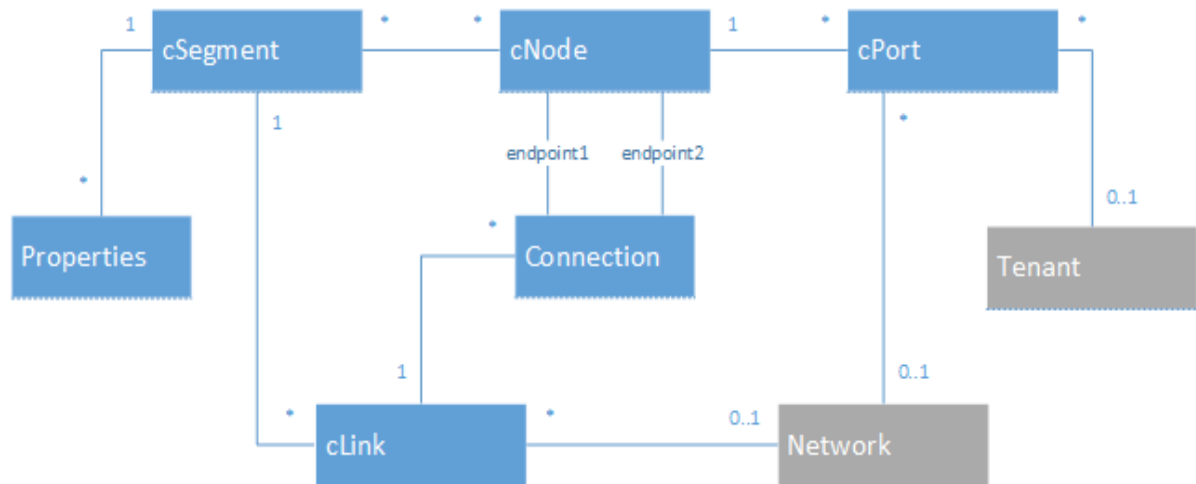


Figure 4 Data model extensions proposed (grey entities already exist)

Configuration files

The necessary configurations are divided among three types of configuration files, what makes the system more generic and the configuration files simpler. It also permits the Device Manager to run segregated from the Network Controller and keep each device independent. Each configuration file is described below.

Network Controller configuration

The main configuration file will provide the necessary configurations for the network controller component. As there are only one Network Controller instance, there should only exists one configuration file in use per deployment.

Device Manager configuration

There should be a configuration file per Device Manager defining how to communicate with the network controller, the list of devices to be managed and the location of the devices' configuration files.

Device configuration

The device manager will load a configuration file per device it is responsible to manage. The file should define the driver used to manage the device and the list of driver specific configurations to be passed to the driver.

This configuration files should all be placed on a common directory and named `[device_name].conf` what will let the Device Manager find them.

API's

This extension proposes four new concepts to introduce on the Neutron API. The new entities and actions are defined in detail below.

A new network type, **campus**, is proposed to identify the new kind of networks presented on this blueprint.

Plugin Interface

As already noticed, the ML2 plugin will be used as a base to implement this extension. It makes no sense to change the ML2 related interfaces with operations specific to this type of network. A new interface will then be created providing the new operations necessary and classes implementing the TypeDriver and MechanismDriver interfaces will also implement this new interface.

The main ML2 plugin will be extended to support the new API defined above through a mixin class that will be provided.

Required Plugin Support

The ML2 plugin will only need to inherit from the new mixin created to implement the new API and add this extension to the supported extensions list.

Dependencies

This extension will be based on the new ML2 framework, so this will be a dependency.

CLI Requirements

The CLI will need a new set of commands to support the new operations provided by the API extension. The full list of new commands is specified below.

| | |
|---|-------------------------------------|
| <code>neutron cnode-create</code> | Create a cNode |
| <code>neutron cnode-delete</code> | Delete a given cNode |
| <code>neutron cnode-update</code> | Update a given cNode |
| <code>neutron cnode-list</code> | List all the cNodes |
| <code>neutron cnode-show</code> | Show details of a given cNode |
| <code>neutron cnode-attach</code> | Attach a cNode to a cSegment |
| <code>neutron cnode-detach</code> | Detach a cNode from a cSegment |
| | |
| <code>neutron csegment-create</code> | Create a cSegment |
| <code>neutron csegment-delete</code> | Delete a given cSegment |
| <code>neutron csegment-update</code> | Update a given cSegment |
| <code>neutron csegment-list</code> | List all cSegments |
| <code>neutron csegment-show</code> | Show details of a cSegment |
| | |
| <code>neutron cport-create</code> | Create a cPort on a cNode |
| <code>neutron cport-delete</code> | Delete a given cPort |
| <code>neutron cport-update</code> | Update a given cPort |
| <code>neutron cport-list</code> | List all the available cPorts |
| <code>neutron cport-show</code> | Show details of a cPort |
| <code>neutron cport-request</code> | Request a cPort for a given tenant |
| <code>neutron cport-release</code> | Release a cPort from a given tenant |
| <code>neutron cport-associate</code> | Assign a cPort to a given network |
| <code>neutron cport-disassociate</code> | Remove a cPort from a network |

| | |
|-----------------------------------|------------------------------|
| <code>neutron clink-create</code> | Create a cLink on a cSegment |
| <code>neutron clink-delete</code> | Delete a given cLink |
| <code>neutron clink-update</code> | Update a given cLink |
| <code>neutron clink-list</code> | List all the cLinks |
| <code>neutron clink-show</code> | Show details of a cLink |

Horizon Requirements

If Horizon will support this extension the following are a list of proposed modifications.

For the `Admin` panel:

- A new item called `Campus Network` under the `System Panel` providing a list of cSegments, cNodes, cPorts and cLinks;
- A new page for each one of the aforementioned items showing their information;
- New menus for create and update each of the components;
- Eventually, a visual representation of the physical network (similar to what currently is available for virtual networks).

For the `Project` panel:

- A new item under `Manage Network` providing a list of cPorts assigned to the tenant;
- If the user has permissions to request cPorts the previous page should also show a list of currently unassigned cPorts;
- The `Network Detail` page should include a list of cPorts belonging to that network.

Usage Example

TBD

Test Cases

TBD

Appendix E

Blueprint

ML2 External Port Extension

ML2 External Port Extension

Scope

The implementation of the Campus Network Extension¹ requires some changes to the ML2 MechanismDriver² interface. The purpose of the current blueprint is to extract a generic interface that can be useful beyond the scope of the aforementioned extension and hopefully reused on future ones.

On this blueprint the concept of external port is introduced with an appropriate API extension. To support the new concept an extension to the MechanismDriver interface is proposed as well as the necessary changes to the current data model.

Use cases

If we get through the Campus Network blueprint we can summarize its final goal as connect non openstack managed hosts to an openstack managed virtual network. Those external hosts can be connected to the network through what will be defined here as external ports.

Providing external ports will let you specify a point of connection to your virtual network so you can for instance:

- Put a couple of switch ports, and the attached hosts, on your virtual L2 network
- Provide an SSID to connect laptops directly to your openstack VMs
- Connect your VMs to a remote location through the internet

All of this can already be done, but the innovation here is that all can be configured automatically and on demand. How that connection can be made is outside the context of this blueprint but in the context of the Campus Network one. The use cases are better explored there also.

Implementation Overview

External Port concept

An external port can be anything that provides a network connection point to non openstack managed hosts, e.g. a router or switch port, a wifi access point, a port on a virtual switch or the endpoint of a VPN tunnel.

The concepts of port and external port differ in a couple of points worth noting:

- External ports don't have associated addresses (neither L2 or L3);
- An external port is not associated with any openstack managed device so the associated device/device-id has a different meaning;
- External ports are usually associated with physical devices what requires a different management approach.

Operation overview

¹ <https://blueprints.launchpad.net/neutron/+spec/campus-network>

² <https://blueprints.launchpad.net/neutron/+spec/ml2-mechanism-drivers>

The main difference between ports and external ports is that ports are usually created and attached to a virtual network on the creation of some resource, either a VM a router or any other, contrary to what should happen with external ports that are the resource per se.

The creation of external ports will then be segregated from its attachment to a virtual network. This will provide a better administration of the resources and enable tenants to quickly migrate their **physical** hosts between different virtual networks.

Creation of an external port

The creation of an external port will follow the next steps:

- The user requests the port creation specifying a device name and a network type
 - The device name will specify the device where the port should be requested
 - The network type will specify which driver will handle the process
- A new port will be requested to the device specified
- If the device has ports available a port identifier will be returned
 - Otherwise the process ends with an error
- The port information will be updated with the port identifier returned
- The port is created

The external ports, unlike ports, will not be associated with a specific segment, but rather with a network type, because on its creation, external ports aren't associated with a specific network, therefore also cannot be associated with a segment.

The port identifier returned by the device is to be used by the user to identify the physical realization of the port created. This can be something like the port number on a switch, the interface name on a router or the SSID of the access point created.

Update of an external port

Update an external port will be very similar to what currently happens with ports. With the only difference that the device may refuse to perform certain actions, case in which the update will fail. A good example is turn a port down (set `admin_state=DOWN`): the device may not be able to turn off a port and may refuse to do that.

Deletion of an external port

As for update, delete an external port will be very similar to what currently happens with ports. The deletion will be requested to the device and the database updated. However, deletion of an external port should first check if the port is associated with a network and disassociate the port first if necessary.

Attach/detach of an external port to/from a virtual network

How the external port will be effectively attached to a virtual network will depend on the network type in use by the external port. This is outside the scope of this blueprint, so we only care here that the MechanismDriver has the appropriate methods for attach and detach an external port to a network.

The Campus Network blueprint proposes a way to implement the attach/detach process, but latter on other approaches can appear. A simple alternative would be the GRE network type permit the creation of tunnels to random devices instead of only among virtual switches.

Data model changes

A new class will be created to store the information of external ports. The data model changes are sketched below on figure 1.

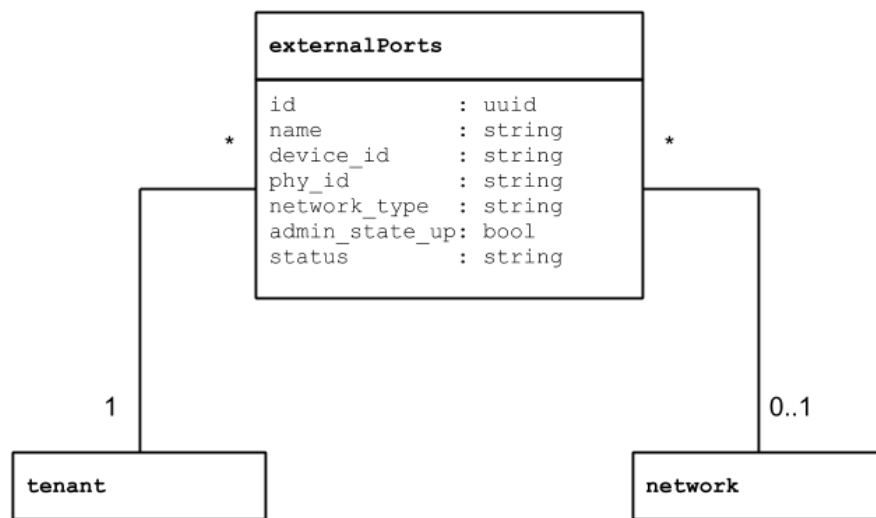


Figure 1 Data model extension (externalPorts)

Configuration files

Changes will certainly be needed by the network type implementations, but as we're only defining the interfaces and operation model, no changes are needed to the configuration files.

API's

Concepts

This extension proposes the new entity **ExternalPort** and a set of operations on top of this new entity. The table 1 below defines the attributes of this new entity. After that the new operations are properly described.

ExternalPort

| Attribute | Type | Required | CRUD | Default value | Validation Constratints | Description |
|----------------|----------|----------|------|---------------|----------------------------|---|
| id | uuid-str | N/A | R | generated | N/A | UUID representing the external port. |
| name | String | no | CRUD | None | No constraints | Human-readable name of the port. |
| device_id | String | yes | CR | N/A | No constraints | Device providing the port. |
| phy_id | String | N/A | R | None | N/A | Port Identifier attributed by the device. |
| network_type | String | yes | CR | N/A | Any available network type | Type of network responsible for the instantiation of this port. |
| admin_state_up | Bool | no | CRU | true | {True False} | Administrative state of the external port. |

| | | | | | | |
|------------|----------|-----|------|------|--|---|
| status | Status | N/A | R | N/A | N/A | Status of the external port: <ul style="list-style-type: none"> • ACTIVE • DOWN • BUILD • ERROR |
| tenant_id | uuid-str | yes | CRU | N/A | No constraint | UUID identifying the tenant holding this external port |
| network_id | uuid-str | no | CRUD | None | Existent network ID belonging to the same tenant as this port. | UUID identifying the network this external port belongs to if it exists |

Table 1 ExternalPort API attributes

Operations

Table 2 summarize the available operations for the external port entity. Each operation is detailed below.

| Operation | Verb | URI | Description |
|-----------|------|-------------------|---|
| Create | POST | /externalports | Requests the creation of an external port on a device. |
| Update | PUT | /externalports/id | Updates an external port. |
| Delete | DEL | /externalports/id | Requests the deletion of an external port from a device. |
| List | GET | /externalports | Lists the available external ports belonging to the tenant. |
| Show | GET | /externalports/id | Shows informations about the external port. |

Table 2 External port operations overview

Create external port

This operation requests the creation of an external port on a given device and associates it with a given tenant.

If the device is able to provide the new port it returns an identifier (**phy_id**) that is then associated with the port. Otherwise the operation fails and the port is not created.

Update external port

The update operation is used to:

- Update the external port name
- Turn the external port up or down (update the **admin_status**)
- Associate the external port to a given tenant
- Attach an external port to a given virtual network
- Detach an external port from a given virtual network

The first and second operations have no special constraints. It's only important to notice that the device can refuse to update the admin status of a port, in which case an error is returned.

The third and fourth operations must ensure that the port is not associated with a network belonging to a tenant different than the one that owns the port.

The last operation has no special constraints.

Delete external port

Request an external port to be deleted from a device. A port cannot be deleted if it is currently attached to a network.

List external ports

List the external ports owned by the tenant that submits the request. According to the policy if the user has admin rights all the available ports are listed.

Show an external port

Returns the information of a specified external port. Only admins or the user that owns the port can see its information.

Plugin interface

The MechanismDriver interface must be extended with new methods to handle the new external port entity. The following list summarizes the new methods to be implemented.

```
create_external_port_precommit(self, context)

create_external_port_postcommit(self, context)

update_external_port_precommit(self, context)

update_external_port_postcommit(self, context)

delete_external_port_precommit(self, context)

delete_external_port_postcommit(self, context)
```

A new context class will also be created for the external ports: `ExternalPortContext`.

Required plugin support

The ML2 plugin will have to implement the new API operations, and call the appropriate MechanismDriver methods.

CLI requirements

The following list of commands are proposed for the `python-neutronclient`.

| | |
|-------------------------------------|---|
| <code>neutron extport-create</code> | Request the creation of an external port on a given device. |
| <code>neutron extport-update</code> | Update the attributes of an external port. |
| <code>neutron extport-delete</code> | Request the deletion of a given external port. |
| <code>neutron extport-list</code> | List the external ports belonging to a tenant. |
| <code>neutron extport-show</code> | Show informations about a given external port. |

References

- [1] Flavio Bonomi et al. «Fog computing and its role in the internet of things». In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12*. MCC '12. New York, New York, USA: ACM Press, 2012, p. 13. ISBN: 9781450315197. DOI: 10.1145/2342509.2342513. URL: <http://dl.acm.org/citation.cfm?doid=2342509.2342513>.
- [2] *Quantum Provider Router Extension*. Visited on September 2013. 2013. URL: <https://blueprints.launchpad.net/neutron/+spec/provider-router>.
- [3] *Campus Network Extension*. Visited on September 2013. 2013. URL: <https://blueprints.launchpad.net/neutron/+spec/campus-network>.
- [4] *ML2 External Port Extension*. Visited on September 2013. 2013. URL: <https://blueprints.launchpad.net/neutron/+spec/ml2-external-port>.
- [5] Timothy Grance; Peter M. Mell; *The NIST Definition of Cloud Computing*. 2011. URL: http://www.nist.gov/manuscript-publication-search.cfm?pub%5C_id=909616.
- [6] Christy Pettey. *Gartner Says Eight of Ten Dollars Enterprises Spend on IT is “Dead Money”*. Visited on August 2013. 2006. URL: <http://www.gartner.com/newsroom/id/497088>.
- [7] *OpenFlow Project*. Visited on September 2013. 2013. URL: <https://www.opennetworking.org/sdn-resources/onf-specifications/openflow>.
- [8] Ronald J Srodawa and Lee A Bates. «An efficient virtual machine implementation». In: *Proceedings of the workshop on virtual computer systems on -*. New York, New York, USA: ACM Press, 1973, pp. 43–73. DOI: 10.1145/800122.803949. URL: <http://portal.acm.org/citation.cfm?doid=800122.803949>.
- [9] Alan Murphy. *Virtualization Defined - Eight Different Ways*. 2007. URL: <http://www.f5.com/pdf/white-papers/virtualization-defined-wp.pdf>.
- [10] Gerald J. Popek and Robert P. Goldberg. «Formal requirements for virtualizable third generation architectures». In: *ACM SIGOPS Operating Systems Review* 7.4 (Oct. 1973), p. 121. ISSN: 01635980. DOI: 10.1145/957195.808061. URL: <http://portal.acm.org/citation.cfm?doid=957195.808061>.
- [11] R Uhlig et al. «Intel virtualization technology». In: *Computer* 38.5 (May 2005), pp. 48–56. ISSN: 0018-9162. DOI: 10.1109/MC.2005.163. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1430631>.
- [12] VMWare. *Understanding Full Virtualization, Paravirtualization, and Hardware Assit*. Tech. rep. VMWare, 2007.
- [13] Wei Chen et al. «A Novel Hardware Assisted Full Virtualization Technique». In: *2008 The 9th International Conference for Young Computer Scientists*. IEEE, Nov. 2008, pp. 1292–1297. DOI: 10.1109/ICYCS.2008.218. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4709159>.
- [14] *OpenStack*. Visited on August 2013. 2013. URL: <http://openstack.org/>.

- [15] *CloudStack*. Visited on August 2013. 2013. URL: <http://cloudstack.org/>.
- [16] *Eucalyptus*. Visited on August 2013. 2013. URL: <http://www.eucalyptus.com/>.
- [17] *Open Nebula*. Visited on August 2013. 2013. URL: <http://opennebula.org/>.
- [18] *Nimbus Project*. Visited on August 2013. 2013. URL: <http://www.nimbusproject.org/>.
- [19] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. «Comparison of multiple IaaS Cloud platform solutions». In: *7th WSEAS International Conference on Computer Engineering and Applications (CEA '13)*. Milan, Italy: WSEAS Press, 2013, pp. 212–217. URL: <http://www.wseas.us/e-library/conferences/2013/Milan/COMPUTERS/COMPUTERS-37.pdf>.
- [20] Gregor von Laszewski et al. «Comparison of Multiple Cloud Frameworks». In: *2012 IEEE Fifth International Conference on Cloud Computing. CLOUD '12*. Washington, DC, USA: IEEE, June 2012, pp. 734–741. ISBN: 978-1-4673-2892-0. DOI: 10.1109/CLOUD.2012.104. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6253573>.
- [21] Junjie Peng et al. «Comparison of Several Cloud Computing Platforms». In: *2009 Second International Symposium on Information Science and Engineering*. IEEE, Dec. 2009, pp. 23–27. ISBN: 978-1-4244-6325-1. DOI: 10.1109/ISISE.2009.94. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5447227>.
- [22] Jiang Zhu et al. «Improving Web Sites Performance Using Edge Servers in Fog Computing Architecture». In: *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*. IEEE, Mar. 2013, pp. 320–323. ISBN: 978-0-7695-4944-6. DOI: 10.1109/SOSE.2013.73. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6525539>.
- [23] *SQLAlchemy*. Visited on September 2013. 2013. URL: <http://www.sqlalchemy.org/>.